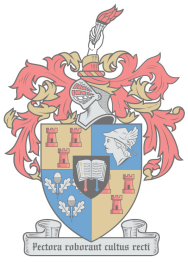
Combining Tree Kernels and Text Embeddings for Plagiarism Detection

by

Jacobus Daniël Thom



Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in the Faculty of Science at Stellenbosch University

Supervisor: Professor A.B. van der Merwe

Co-supervisor: Doctor R.S. Kroon

March 2018

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: March 2018 .....

Abstract

The internet allows for vast amounts of information to be accessed with ease. Consequently, it becomes much easier to plagiarize any of this information as well. Most plagiarism detection techniques rely on n-grams to find similarities between suspicious documents and possible sources. N-grams, due to their simplicity, do not make full use of all the syntactic and semantic information contained in sentences. We therefore investigated two methods, namely tree kernels applied to the parse trees of sentences and text embeddings, to utilize more syntactic and semantic information respectively. A plagiarism detector was developed using these techniques and its effectiveness was tested on the PAN 2009 and 2011 external plagiarism corpora. The detector achieved results that were on par with the state of the art for both PAN 2009 and PAN 2011. This indicates that the combination of tree kernel and text embedding techniques is a viable method of plagiarism detection.

Opsomming

Die internet laat mens toe om groot hoeveelhede inligting maklik in die hande te kry. Gevolglik word dit ook baie makliker om plagiaat op enige van hierdie inligting te pleeg. Meeste plagiaatopsporingstegnieke maak staat op n-gramme om ooreenkomste tussen verdagte dokumente en moontlike bronne op te spoor. Aangesien n-gramme taamlik eenvoudig is, maak hulle nie volle gebruik van al die syntaktiese en semantiese inligting wat sinne bevat nie. Ons ondersoek dus twee metodes, naamlik boomkernfunksies, wat toegepas word op die ontledingsbome van sinne, en teksinbeddings, om onderskeidelik meer sintaktiese en semantiese inligting te gebruik. 'n Plagiaatdetektor is ontwikkel met behulp van hierdie twee tegnieke en die effektiwiteit daarvan is getoets op die PAN 2009 en 2011 eksterne plagiaatkorpora. Die detektor het resultate behaal wat vergelykbaar was met die beste vir beide PAN 2009 en PAN 2011. Dit dui aan dat die kombinasie van boomkern- en teksinbeddingstegnieke 'n redelike metode van plagiaatopsporing is.

Acknowledgements

I would like to thank the CSIR-SU Centre for Artificial Intelligence Research for their generous financial support during this study.

Contents

List of Figures

List of Tables

Introduction

While there is no single, universally agreed upon definition, the essence of plagiarism comes down to two things:

1. using the work and ideas of others without acknowledgement,
2. while representing it as originating from oneself.

Definitions start to diverge when it comes to how and when something may be used freely, as well as how credit should be given where it is due. For an extreme example, one does not need to provide a citation after each and every English word.

The reason plagiarism is bad, therefore, is that in the definition above, clause one constitutes theft, while clause two constitutes fraud. Additionally, from an academic perspective, using someone else's work without extending/building on it means no personal growth has been made as a scholar or researcher, and the pool of human knowledge has simply been diluted.

Plagiarism, while a serious issue, has in the past been curtailed by the relative difficulty of performing it. A thousand or more years ago, one probably had to see someone do something in person, or physically acquire an object oneself in order to copy it. The advent of books and their widespread availability meant one (only) had to acquire a book on the required topic (if it existed) and read about it. With the rise of the internet and its spread to more and more devices, finding information on virtually any topic is as easy as opening a browser on your smartphone and performing a (conveniently automated) search using some key words. There is no doubt that the amount of information available as well as how easy it is to access will only increase.

Both the volume of information and the ease with which one can copy/plagiarize it, necessitate the creation of automated plagiarism detection tools.

Having established the need for plagiarism detection, the question of how to perform such detection arises. By its very nature, one cannot simply plagiarize, one must plagiarize something. This means that, at its core, plagiarism detection consists of a comparison between some work deemed suspicious of being plagiarism and some work(s) deemed to be the potential source(s) thereof. For a suspicious work to plagiarize a source work, the two must be similar. Automated plagiarism detection must therefore use measures which quantify similarity and make decisions based on this.

For this thesis, we are considering (English) text plagiarism exclusively. If one ignores whether or not credit was given, one piece of text can be said to plagiarize another if they have the same meaning – in other words, they share semantic similarity. In order to quantify this similarity, two techniques will be investigated – the one being tree kernels and the other text embeddings.

Objectives

The work done for this thesis had three broad objectives. The first of these was to develop a plagiarism detector that rivaled or surpassed the state of the art on the PAN (Plagiarism Analysis, Authorship Identification, and Near-Duplicate Detection) 2009 and 2011 competitions' corpora. The second was to combine the use of tree kernels over parse trees – to utilize syntactic information – and text embeddings – to incorporate semantic information – in a way that produced accurate similarity scores for sentence pairs. These similarity scores form the basis for plagiarism detection in this thesis and therefore also determine to a large extent how well the first objective is achieved. The last was to try and understand, as far as possible, why the techniques that worked did so, and why others did not.

The PAN corpora were chosen, because they are fairly large and provide annotations for both a training and a test set. This makes them well suited to performing and evaluating automated plagiarism detection on. After 2011, however, the competitions shifted their focus to incorporating online searching as part of the detection process. This falls outside the scope of the thesis and, as such, the 2011 corpus is the last one considered. Furthermore, the 2010 corpus is mostly a much larger version of the 2009 corpus and is also not investigated.

A brief look at the methods used in the above PAN competitions [1, 2, 3] reveals that n-gram based methods are preferred, almost exclusively. This is likely due to their simplicity and speed. However, exactly due to this simplicity, n-grams are not very efficient at using all the information (syntactic and semantic) contained in a sentence. Turning to tree kernels over the parse trees of sentences and various levels of text embeddings is, therefore, an attempt to see if utilizing more of the information can lead to improved plagiarism detection.

Thesis Overview

The thesis is divided into three chapters, excluding this introduction and the conclusion.

The first of these chapters covers topics that constitute background knowledge and other information useful for understanding the various techniques used. Since a core theme in this thesis is the use of tree kernels and text embeddings, each of these two topics forms a section in this chapter. The section on feature vectors is first and touches on how vectors (embeddings) for words and larger pieces of text are constructed using the Word2Vec and Doc2Vec methods respectively.

Between the section on feature vectors and the section on tree kernels, there is a short section on the parsing of text. As such, it is useful as a lead-in to the next section, since the tree kernels discussed there will operate on the tree structures generated during the parsing of text.

The tree kernel section gives an overview of three different kinds of kernels and how they are used to obtain a similarity score from the parse trees of two sentences.

The heart of plagiarism detection approaches used in this thesis, involves comparing many sentences to many others and calculating a score for each pair based on their similarity. One key assumption that was made, is that only one suspicious sentence can plagiarize any one source sentence (mostly to reduce the granularity of detections, see Section 1.5.1). To find the best (most similar overall) set of sentence pairs, the Hungarian algorithm is used. How the Hungarian algorithm works, is described the section after the one on tree kernels.

Many of the choices made throughout this thesis can be traced back in some form to how well they work in the context of the PAN corpora. In light of this, there

is next a section which gives more detail on how these corpora are constructed.

Finally there is a section that deals with related work in the field of plagiarism detection. In particular, it discusses the approaches used by others to perform detections – specifically those of the entrants to the PAN competitions.

The second chapter describes how our approach to plagiarism detection is performed in detail.

It starts with a section on how text is (pre-)processed into the various forms that the detector requires.

Since performing a similarity analysis between two texts – this is the basis for a detection – is computationally expensive, suspicious documents are first compared with source documents in a very coarse grained fashion to find the most likely areas of plagiarism for finer scrutiny. An exposition of how this is done, follows after the pre-processing section.

Next is a section explaining the inner workings of the actual plagiarism detection step. A core part of this is the various classifiers that calculate the similarities of sentence pairs. The bulk of this section is therefore dedicated to describing these classifiers.

The final step of the plagiarism detector merges suspicious/source sentence pairs into larger suspicious/source passage pairs. Each of these passage pairs becomes an annotation and these annotations are what the user receives as output from running the detector. How, when and why sentences are merged is explained in a section on post-processing.

The third chapter presents the results of applying the detector to the PAN 2009 and 2011 corpora as well as how it compares to the other detectors submitted for those competitions. The ensuing discussion of these results aims to give insight into how the constituent parts of the detector affect them.

Finally, the conclusion provides some perspective on the thesis as a whole.

Chapter 1

Background

This chapter is dedicated to the core concepts and techniques used in the rest of this thesis and the work leading up to it.

1.1 Feature Vectors

If one wants to compare many pieces of text with many others – as one needs to do when checking for plagiarism – one encounters a problem: how does one quantify the similarity of any given pair of texts? An elegant solution comes in the form of feature vectors. A feature vector is simply an n-dimensional vector of numbers that somehow describes or encapsulates the (important) properties of an object. Once one has constructed a feature vector for each object, one can draw on the existing methods for comparing vectors. The similarity score obtained by comparing the two feature vectors is then used as the similarity score for the original object pair. One widely-used similarity measure employed throughout this thesis is called cosine similarity. If $\mathbf{a}$ and $\mathbf{b}$ are the two vectors in question, then given that the usual definition of the inner product is,

$$\mathbf{a} \cdot \mathbf{b} \equiv \sum_{i=1}^{n} a_i b_i \,, \tag{1.1}$$

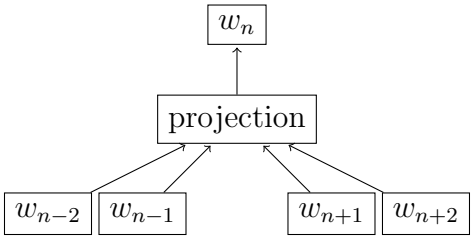


Figure 1.1: CBOW configuration.

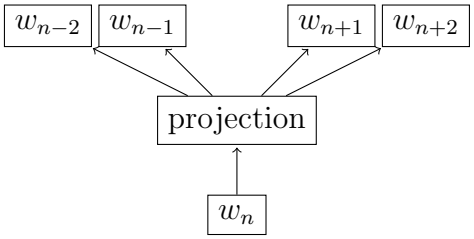


Figure 1.2: Skip-gram configuration.

where a_i and b_i are the i'th component of $\mathbf{a}$ and $\mathbf{b}$ respectively, and the norm of a vector is,

$$\|\mathbf{a}\| = \sqrt{\mathbf{a} \cdot \mathbf{a}}\,, \tag{1.2}$$

then the cosine similarity of $\mathbf{a}$ and $\mathbf{b}$ is defined as,

$$\text{cosine similarity}(\mathbf{a}, \mathbf{b}) \equiv \cos(\theta) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}\,, \tag{1.3}$$

where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$.

Using feature vectors shifts the problem of quantifying the similarity of two text passages to finding good feature vectors for these passages. Methods for finding such vectors for words and longer pieces of text are described in the following sections.

1.1.1 Word2Vec

Although there exist many different ways for constructing a vector for a given word (e.g. neural network language models [4], latent semantic analysis [5], latent Dirichlet allocation [6]), one of the most popular is a recent method known as Word2Vec [7]. Word2Vec uses a neural network to construct a vector for a word by using the context the word is found in.

Neural Networks A small digression on one of the simplest kinds of neural networks, namely feed forward networks, is as follows. Consider a directed graph with nodes organized in layers. Nodes in the graph typically have incoming connections from all the nodes in the previous layer and outgoing connec-

tions to all nodes in the next layer. At each node, the weighted sum of all its inputs is calculated and passed on as output. Often this weighted sum is passed through a (typically non-linear) function before the result of that is passed on. A neural network is, therefore, essentially a function that calculates a result by transforming input according to rules at each node and how the nodes are connected by the abovementioned graph. If one wants to train a neural network to perform a specific kind of calculation, one typically creates a set of examples consisting of an input and an expected outcome for that input. Training a neural network then consists of giving it these inputs at the first layer and examining the output at the final layer. By using the difference between the seen output and the expected output for a given input, one can change the weights inside the network so that the seen output more closely mimics the desired one. The most common way to do this is by using a method called backpropagation. This process is repeated many times with all the examples in order to obtain a set of weights that gives good results. A good introductory text on neural networks with much more detail on everything mentioned here can be found in [8, 9].

In general there are two main configurations of the neural network used by Word2Vec [7]. The one is called the continuous bag of words (CBOW) model (Figure 1.1) and the other is called the skip-gram model (Figure 1.2). In both figures, the input layer is at the bottom and the output is at the top. The figures are stylized representations of the actual architectures in order to highlight the differences between the two.

These Word2Vec neural networks typically do not take the words themselves as input, but rather the so-called 'one-hot' encoding of the words. A 'one-hot' encoding is a vector with zeroes everywhere except at a single position, which is set to 1. Consider a corpus with only three different words making up all the text: 'the', 'cat' and 'sat'. The vocabulary size for this corpus is then three. The dimension of 'one-hot' vectors is the size of the vocabulary and the position of the 1 indicates the specific word in the vocabulary. For the corpus just described, one might have that 'cat' is encoded by $(1, 0, 0)$, 'sat' by $(0, 1, 0)$ and 'the' by $(0, 0, 1)$. In the discussion that follows, when there are references to 'words' as input or output

of a Word2Vec network, these 'one-hot' vectors are meant.

If one takes w_n to be the n'th word in a piece of text, then the context of w_n is generally defined as the m words preceding and following w_n. In other words, with a context window size of m, the context of w_n are the words w_i where i ranges from $n - m$ to $n + m$ and $i \neq n$. In the case of the CBOW configuration, the context words are the *input* to the neural network. What the network tries to predict as its output then is a word w_n, given its context.

The skip-gram configuration can be seen as the CBOW configuration turned on its head. Here, the network receives only the word w_n as input and tries to predict a context for it. The word 'a' is used here, because in practice the context is not always chosen as just the m preceding and m following words. Words can be skipped (hence the name skip-gram) and there need not be exactly the same number of context words either side of w_n. Word order is still preserved. This allows for many contexts to be generated, which in turn improves the quality of vectors generated for words that only appear infrequently or for small corpora in general. This does, of course, mean that skip-gram training typically takes longer than CBOW training, assuming the same rate of convergence.

In both cases the layer labeled `projection` (in Figures 1.1 and 1.2) is responsible for storing the weights that become the word embeddings once training is complete. For example, if a corpus has a vocabulary with a million words and one is using Word2Vec to train embeddings of size 300, then the weights would form a matrix of size 1000000×300 – one row vector of size 300 for each word. In the case of skip-gram, a word is mapped to its vector in this (projection) layer by looking up the row corresponding to the word. This vector is the output of this layer. For CBOW, the same mapping is performed, however, all the resulting vectors (one for each word in the input) are typically summed together or averaged before becoming the final output of this layer.

1.1.2 Doc2Vec

The paper that described the Word2Vec method [7] was followed a year later by one on what the authors call paragraph vectors [10]. The `gensim` [11] library we use for training paragraph vectors calls this method Doc2Vec, and as such we shall

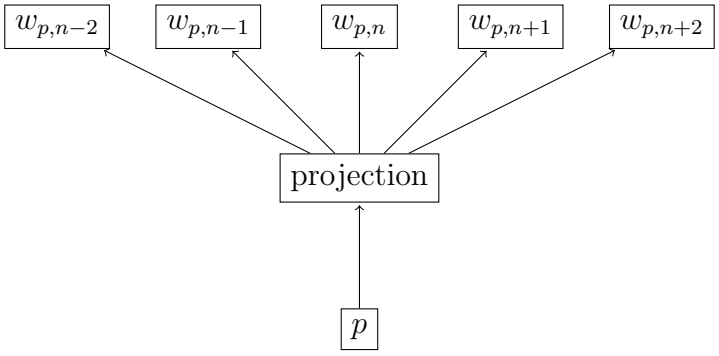


Figure 1.3: Distributed Bag-of-Words (DBOW) configuration.

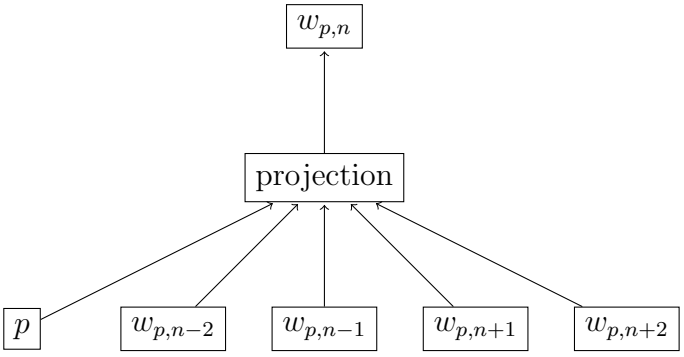


Figure 1.4: Distributed Memory (DM) configuration.

refer to it by this name henceforth.

Whereas Word2Vec is a way for finding an embedding for a word, Doc2Vec aims to do so for longer passages of text. The way in which Doc2Vec does this, however, is surprisingly simple: it treats a piece of text as a special kind of word. In order to elaborate on this, consider the two architectures (Figures 1.3 and 1.4) for Doc2Vec, which are very similar to those of Word2Vec. Somewhat confusingly, the Doc2Vec analogue of the skip-gram model is called Distributed Bag-of-Words (DBOW) (Figure 1.3) and the analogue of CBOW is called Distributed Memory (DM) (Figure 1.4). In these figures, p is a unique identifier for a passage (or document) and $w_{p,n}$ refers to the n'th word in passage p. For the DM model, one therefore adds the passage's identifier to each context coming from that specific passage. Instead of attempting to predict a context given a word as the skip-gram model does, the DBOW model aims to predict a context given a passage. In this case a context is a sequence of words coming from the passage by randomly sampling

a text window. From this short description, the reasoning behind the chosen names becomes clearer. A paragraph vector in the Distributed Memory configuration acts as an object which remembers which words it was trained with [10]. For the Distributed Bag-of-Words configuration on the other hand, it acts as the glue which binds many smaller Bags-of-Words together.

There is one thing to note here. When in the DM configuration, the network trains the word vectors and paragraph (or document) vector at the same time, but the word vectors are shared between paragraphs. For DBOW, however, one only trains the vectors for the paragraphs and consequently the model is smaller.

1.2 Parsing of Text

In the context of this thesis, tree kernels (described in Section 1.3) are applied to parse trees of sentences. This section, therefore, briefly discusses two kinds of parsing and the resulting parse trees that we consider.

1.2.1 Constituency Parse Trees

There are two main ways to parse a sentence into a tree. One of these is called constituency parsing. An example of a constituency parse tree for the sentence, 'The cat sat on the mat.', can be seen in Figure 1.5. In this example, punctuation is treated as a separate token and therefore has its own node. Constituency parsing aims to convert a sentence into a tree, by dividing it into phrases. In the broadest sense, phrases can be either noun phrases (NP) or verb phrases (VP). As such, the rules that govern this kind of parsing are known as phrase structure grammars [12]. Phrases can be divided further into more phrases or the part-of-speech (PoS) (such as nouns, verbs, etc.) of a single word. A node that represents such a PoS then has the actual word in the sentence as a child. In a constituency tree, therefore, the leaves are the words themselves.

The specific rule (in the grammar) represented by a node and its direct children is sometimes called the production associated with the node. For example, (S $\rightarrow$ NP, VP) is the production of the root of the tree shown in Figure 1.5, and means that the sentence (S) is divided into a noun phrase (NP) and a verb phrase (VP).

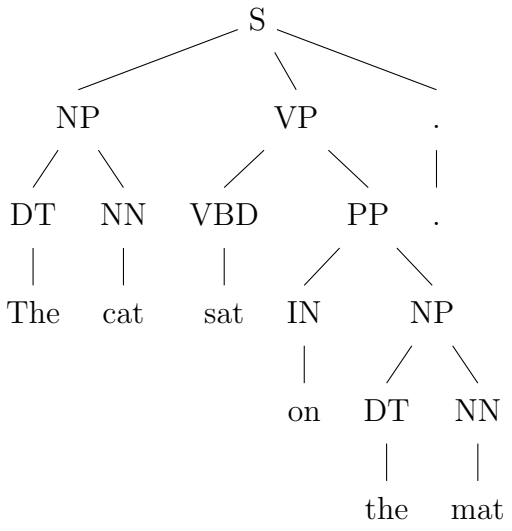


Figure 1.5: Constituency parse tree of the sentence: 'The cat sat on the mat.'

1.2.2 Dependency Parse Trees

Dependency parsing is the other main method of parsing and the rules for dependency relationships [13] were developed at around the same time as those of phrase structure grammars. Figure 1.6 shows the dependency parse tree for the same sentence used before, namely 'The cat sat on the mat.'. Whereas a constituency parse tree breaks a sentence up into smaller and smaller phrases until one reaches a single word, a dependency parse tree only contains nodes for the actual words in the sentence. It is, therefore, different from a constituency tree in that there are no phrasal nodes and that the structure of the tree itself depicts the relationships (dependencies) between the elements of a sentence. This means that a dependency tree is smaller than a constituency tree of the same sentence. As such, one of the reasons we use dependency trees throughout the rest of this thesis is due to this reduced size (which provides memory and speed benefits).

1.3 Tree Kernels

When comparing two items algorithmically, the (normalized) inner product (i.e., the cosine similarity) between the feature vectors of the items is often used. For

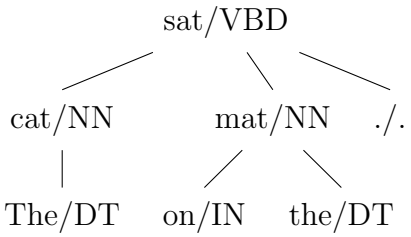


Figure 1.6: Dependency parse tree of the sentence: 'The cat sat on the mat.'

tree structures – while constructing such a feature vector is by no means impossible – the dimensionality of the space in which these vectors are embedded can be intractably large [14]. In cases such as these, it is usually far more efficient to use a (tree) kernel to calculate an inner product: the kernel function provides a similarity measure on some implicit feature space without having to first calculate the vectors explicitly [14, 15]. Strictly speaking, in order to obtain a similarity measure (value between 0 and 1), one needs to normalize the kernel function, since without normalization the kernel function is equivalent to an inner product. In the case of tree kernels, for example, if T_1 and T_2 are trees and $\mathrm{TK}(T_1, T_2)$ is a tree kernel acting on them, then one can normalize by calculating

$$\mathrm{TK}_*(T_1, T_2) \equiv \frac{\mathrm{TK}(T_1, T_2)}{\sqrt{\mathrm{TK}(T_1, T_1) \times \mathrm{TK}(T_2, T_2)}}, \tag{1.4}$$

where $\mathrm{TK}_*(\cdot, \cdot)$ is the normalized tree kernel. This normalization method is used for all the tree kernels that follow.

One can define different kinds of tree kernels based on the kinds of sub-structure fragments (within the trees) that are considered in the actual kernel computation. This means that different kernels correspond to different vector space embeddings. For example, one kind of tree kernel, called the subtree kernel (STK), considers only full sub-trees (a node and all its descendants) in the kernel function. In general, the tree kernel is a function over all the pairs of nodes between two trees – specifically the pair of nodes and the sub-structures rooted at them. If T_1 and T_2 are trees

then the kernel function is simply [14],

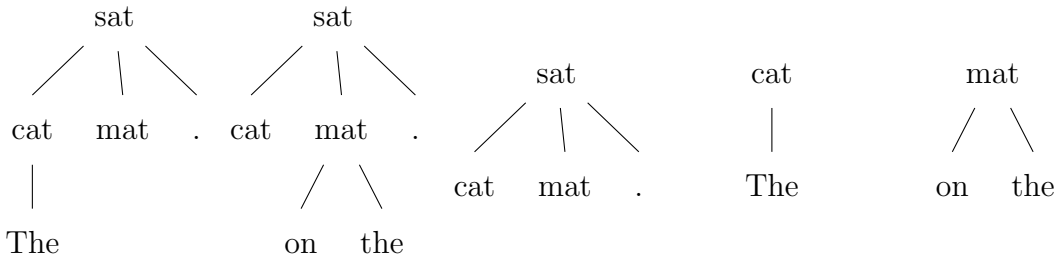
$$\mathrm{TK}(T_1, T_2) \equiv \sum_{n_1 \in T_1} \sum_{n_2 \in T_2} \Delta(n_1, n_2)\,, \tag{1.5}$$

where exact definition of the function $\Delta(n_1, n_2)$ varies depending on the type of tree kernel. Due to the structure of trees, the Δ's can usually be defined very concisely in a recursive fashion. This is the case for all the kernels listed below.

The following sections describe the tree kernels that were used in the plagiarism detector that was developed.

1.3.1 Subset Tree Kernel

Subset tree kernels (SSTKs) are different from STKs in that the fragments (i.e., the subtree structures) considered need not extend all the way to the leaves. However, if one child of a node is included in a particular fragment then all other children must be as well. For the example in Figure 1.6, the fragment rooted at 'sat' together with its three children, 'cat', 'mat' and '.', would be a subset tree, but not a subtree. In particular, all the subset trees (excluding single nodes) of Figure 1.6 are:



The Δ-function corresponding to this particular choice of tree fragment [14] follows below. Let us consider two trees T_1 and T_2 with n_1 a node of T_1 and n_2 a node of T_2. Furthermore, define a 'pre-terminal' node as one that only has leaves as children. Then the Δ-function is given by:

1. If n_1 and n_2 are different, then $\Delta(n_1, n_2) = 0$.
2. If n_1 and n_2 are the same and
   (a) n_1 and n_2 are pre-terminals, then $\Delta(n_1, n_2) = \lambda$.

(b) n_1 and n_2 are NOT pre-terminals, then
$\Delta(n_1, n_2) = \lambda \prod_{i=1}^{\min(|n_1|,|n_2|)} (1 + \Delta(c_i^1, c_i^2))$.

Here $|\cdot|$ indicates the number of children of the particular node and c_i^1 is the i'th child of node n_1.

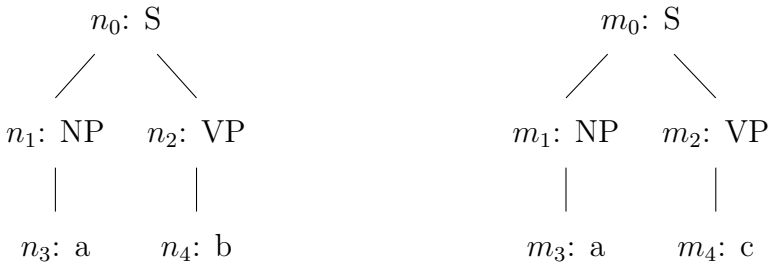
An important note on the above is that SSTKs were originally designed with constituency trees in mind. In the definition of the Δ-function above then, n_1 and n_2 are the same when their productions are the same. As mentioned in Section 1.2.1, a production, from the general view point of tree structures, is a node and all its (direct) children.

Before going into what the λ parameter is for, consider the case when $\lambda = 1$. In this case $\Delta(n_1, n_2)$ counts the number of common tree fragments between the sub-trees rooted at n_1 and n_2. To see this, consider the recursive definition given above. The first two cases (1 and 2a) are trivial. For the third case (2b), one can form a common fragment by taking the current node pair (n_1, n_2) together with any of the common fragments rooted at common children. This gives $1 + \Delta(c_i^1, c_i^2)$ possibilities at the i'th child pair.

When the two trees given to the tree kernel are exactly the same, the number of common fragments can number in the thousands or even millions [14]. For differing trees, this can easily be orders of magnitude less. With $\lambda = 1$ therefore, one is in the situation where the kernel function is very strongly peaked around trees being exactly the same. To smooth things out and reduce the effect of larger tree fragments dominating the kernel (many smaller sub-fragments contribute to the larger ones), one can apply a weighting factor to every fragment. This is what λ does. For every increase in fragment height, that fragment has its contribution weighted by an additional λ-factor. By choosing $0 < \lambda \leq 1$, fragments become exponentially down-weighted in their size.

The worst case run time complexity of the SSTK is $O(\rho^3 |T_1||T_2|)$ [16], where ρ is the maximum branching factor found in either of trees T_1 and T_2. However, by sorting the trees according to the production at the nodes, one can find those node pairs which will result in non-zero contributions in $O(n_1 \log(n_1) + n_2 \log(n_2))$. After this pre-processing step, the average run time is much closer to linear [16].

Example 1.1. At this point, a small worked example might provide more clarification. Consider two constituency trees N and M, with nodes $n_i \in N$ and $m_i \in M$ respectively. Suppose the contents of the two trees are as shown below, where the format in each node is 'node identifier: node label'.



To calculate the SSTK for these trees, let us start from the definition (1.5):

$$\mathrm{SSTK}(N, M) = \sum_{n \in N} \sum_{m \in M} \Delta(n, m) \tag{1.6}$$

Looking at trees N and M, the only pairs of nodes with the same production are (n_0, m_0) and (n_1, m_1). The pair (n_2, m_2) does not have the same production, because of the difference in the respective child node labels: 'b' versus 'c'. This means that all other pairs (e.g., (n_0, m_1) or (n_0, m_2)) make zero contribution to the tree kernel according to Rule 1 in the defintion of the SSTK's Δ-function. Therefore,

$$\mathrm{SSTK}(N, M) = \Delta(n_0, m_0) + \Delta(n_1, m_1)\,. \tag{1.7}$$

From Rule 2a of the Δ-function,

$$\Delta(n_1, m_1) = \lambda, \tag{1.8}$$

since both n_1 and m_1 are pre-terminal nodes. For $\Delta(n_0, m_0)$ one needs to make use

of Rule 2b of the Δ-function:

$$\Delta(n_0, m_0) = \lambda \prod_{i=1}^{2} \left(1 + \Delta(c_i^N, c_i^M)\right) \tag{1.9}$$

$$= \lambda \left(1 + \Delta(n_1, m_1)\right) \left(1 + \Delta(n_2, m_2)\right) \tag{1.10}$$

$$= \lambda(1 + \lambda)(1 + 0) \tag{1.11}$$

$$= \lambda + \lambda^2 \,. \tag{1.12}$$

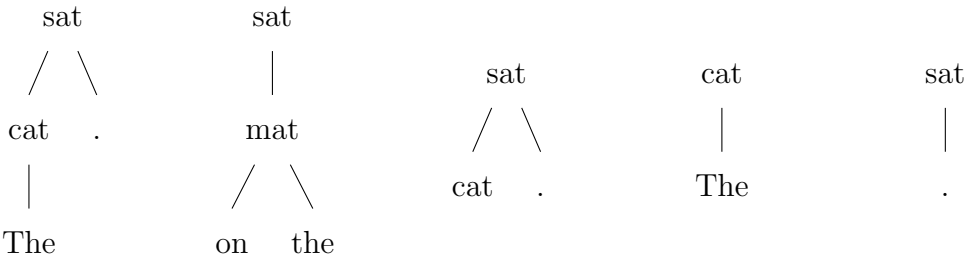
Therefore, one finds that

$$\text{SSTK}(N, M) = \lambda + \lambda^2 + \lambda \tag{1.13}$$

$$= 2\lambda + \lambda^2 \,. \tag{1.14}$$

1.3.2 Partial Tree Kernel

Just as subset tree kernels generalize sub-tree kernels, so too do partial tree kernels generalize subset tree kernels. If one drops the condition that all sibling nodes must be included if one sibling is – used when creating subset tree fragments – the resulting fragments are called partial tree fragments. Partial tree kernels (PTKs), as the name implies, operates on this type of fragment and were introduced [17] to take greater advantage of the structure of dependency trees. Some partial trees from the example dependency tree in Figure 1.6 are:



It should be clear from the definition of partial trees that there are many more partial tree fragments than subset tree fragments in all but the simplest of trees.

The Δ-function for the PTK [17] is somewhat more complicated than that of the SSTK. Consider once again two trees T_1 and T_2 with n_1 a node of T_1 and n_2 a node of T_2, then the Δ-function is defined as:

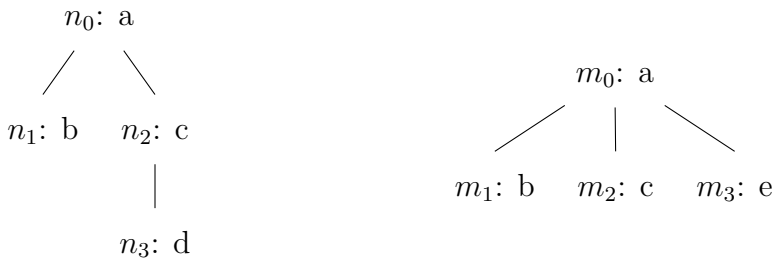
1. If the node labels of n_1 and n_2 are different, then $\Delta(n_1, n_2) = 0$.
2. If the node labels of n_1 and n_2 are the same, and
   (a) either n_1 or n_2 is a leaf, then $\Delta(n_1, n_2) = \lambda\mu^2$,
   (b) otherwise
   $\Delta(n_1, n_2) = \lambda\left(\mu^2 + \sum_{I_1,I_2,\ell(I_1)=\ell(I_2)} \mu^{d(I_1)+d(I_2)} \prod_{k=1}^{\ell(I_1)} \Delta(c_{i_{1k}}, c_{i_{2k}})\right).$

$I_1 = i_{11}, i_{12}, i_{13}, \ldots, \ell(I_1)$ and $I_2 = i_{21}, i_{22}, i_{23}, \ldots, \ell(I_2)$ are sequences of indices of the child nodes of n_1 and n_2 respectively with the same (finite) length, up to the minimum of either the number of children of n_1 or n_2. In the above, the function $\ell(I)$ gives the length of sequence I and $d(I_j) = i_{j\ell(I_j)} - i_{j1}$ (i.e., the difference between the first and last index in sequence I_j). Furthermore, $c_{i_{jk}}$ is the child of n_j at index i_{jk}. It is worth noting that for the SSTK the recursion proceeds until a pre-terminal is reached, whereas for PTK the recursion stops at the leaves.

The factor λ performs the same function as it did for the SSTK in Section 1.3.1. The new factor μ also performs down-weighting, but this time on the length of child sequences [17]. Just as for λ, μ takes on values larger than 0 and no more than 1. For the SSTK, it was mentioned that when $\lambda = 1$, $\Delta(n_1, n_2)$ simply counts the number of common fragments between the sub-trees rooted at n_1 and n_2. The same holds true here when both $\lambda = 1$ and $\mu = 1$.

The run time complexity for the PTK is exactly the same as for the SSTK: $O(\rho^3|T_1||T_2|)$ in the worst case and close to linear on average [16, 17, 18]. The only change that needs to be made to the discussion given in Section 1.3.1 is that one sorts by the label of the node, instead of its production.

Example 1.2. For the PTK, let us look at an example calculation using dependency trees. Suppose two such trees are N and M (given below) with nodes n_i and m_i respectively. As in the SSTK example, the format of the nodes is 'node identifier: node label'.



Once again, one starts with the definition (1.5):

$$\mathrm{PTK}(N, M) = \sum_{n \in N} \sum_{m \in M} \Delta(n, m)\,. \tag{1.15}$$

By applying Rule 1 of the PTK Δ-function, it is clear that only three terms contribute to the PTK, so that

$$\mathrm{PTK}(N, M) = \Delta(n_0, m_0) + \Delta(n_1, m_1) + \Delta(n_2, m_2)\,. \tag{1.16}$$

In the case of $\Delta(n_1, m_1)$, both n_1 and m_1 are leaves. For $\Delta(n_2, m_2)$, m_2 is a leaf. Both of these cases are therefore captured by Rule 2a of the Δ-function and contribute a factor of $\lambda\mu^2$:

$$\Delta(n_1, m_1) = \Delta(n_2, m_2) = \lambda\mu^2\,. \tag{1.17}$$

For the final term, $\Delta(n_0, m_0)$ one needs to make use of Rule 2b. Since n_0 has fewer children (than m_0), namely 2, sequences considered can maximally be of this length. Possible sequences, I_N, of the children of n_0 are therefore, (n_1), (n_2) or (n_1, n_2). Similarly, for m_0 the possibilities for I_M are (m_1), (m_2), (m_3), (m_1, m_2), (m_1, m_3) or (m_2, m_3). This leads to,

$$\Delta(n_0, m_0) = \lambda \left[\mu^2 + \sum_{I_N, I_M, \ell(I_N) = \ell(I_M)} \mu^{d(I_N) + d(I_M)} \prod_{k=1}^{\ell(I_N)} \Delta(c^N_{i_{1k}}, c^M_{i_{2k}}) \right] \tag{1.18}$$

$$= \lambda \left[\mu^2 + \mu^{0+0}(\Delta_{11} + \Delta_{12} + \Delta_{13} + \Delta_{21} + \Delta_{22} + \Delta_{23}) \right. \tag{1.19}$$

$$\left. + \mu^{1+1}(\Delta_{11}\Delta_{22} + \Delta_{11}\Delta_{23} + \Delta_{12}\Delta_{23}) \right]\,, \tag{1.20}$$

where Δ_{ij} is used as shorthand for $\Delta(n_i, m_j)$. Many terms in the above are, of

course, zero. This simplifies $\Delta(n_0, m_0)$ to,

$$\Delta(n_0, m_0) = \lambda \left[\mu^2 + (\lambda\mu^2 + \lambda\mu^2) + \mu^2(\lambda^2\mu^4)\right] \tag{1.21}$$
$$= \lambda\mu^2 + 2\lambda^2\mu^2 + \lambda^3\mu^6 \,. \tag{1.22}$$

Putting everything together gives the final result as,

$$\mathrm{PTK}(N, M) = 3\lambda\mu^2 + 2\lambda^2\mu^2 + \lambda^3\mu^6 \,. \tag{1.23}$$

1.3.3 Smoothed Partial Tree Kernel

Smoothed partial tree kernels [18] (SPTKs) are PTKs with one important difference. Whereas for PTKs node labels have to match exactly – otherwise the two tree fragments rooted at such nodes make no contribution to the PTK value – for SPTKs the two fragments still contribute, but weighted using a chosen similarity function for comparing the node labels. In theory, this should improve SPTK's ability to compare sentences with similar meanings and layout (semantics and syntax), but different word choices.

Consider for example two partial tree fragments being compared, one rooted with a node labeled a and one with a node labeled b. In the case of PTK, since $a \neq b$, these two fragments add nothing to the overall value calculated by the kernel. For SPTK, however, one calculates the similarity value between a and b (say $\mathrm{sim}(a, b)$), and proceeds as one would for PTK if a were equal to b, finally multiplying the Δ-function contribution for this node pair by $\mathrm{sim}(a, b)$.

To make this discussion more precise, the definition of the recursive part of the Δ-function for the SPTK is simply:

$$\Delta(n_1, n_2) = \mathrm{sim}(a, b)\lambda \left(\mu^2 + \sum_{I_1, I_2, \ell(I_1)=\ell(I_2)} \mu^{d(I_1)+d(I_2)} \prod_{k=1}^{\ell(I_1)} \Delta(c_{i_{1k}}, c_{i_{2k}}) \right) , \tag{1.24}$$

where a is the node label of n_1 and b is the node label of n_2. If either node is a leaf, then this works out to be $\mathrm{sim}(a, b)\lambda\mu^2$. Rule 1 of the PTK Δ-function is, therefore,

not explicitly used any more, but its effect may still occur when $\text{sim}(a, b) = 0$.

The worst case run time complexity for the SPTK is, once again, the same as for PTK and SSTK [16, 17, 18]. However, due to the use of similarity scores between node labels instead of exact matching, there is no convenient pre-processing step that can be taken to reduce the time complexity in general.

A full worked example for SPTK would be quite lengthy even for the small trees used in the PTK example, because all pairs of nodes now typically have non-zero contribution. However, the basic steps that occur during the evaluation of an SPTK is exactly the same as for PTK, with one modification. For PTK, when calculating the contribution of a pair of nodes via $\Delta_{\text{PTK}}(n_1, n_2)$, one checks if the node labels of n_1 and n_2 are the same or not. If one instead ignores this check and relies exclusively on the factor $\text{sim}(a, b)$ (where a, b is the node label of n_1, n_2 respectively) then the contribution for this pair in the case of SPTK becomes $\Delta_{\text{SPTK}}(n_1, n_2) = \text{sim}(a, b)\Delta_{\text{PTK}}(n_1, n_2)$. In other words, if one performs a PTK calculation, while ignoring the same/different condition of the node labels, then the corresponding SPTK value can be found by replacing every $\Delta_{\text{PTK}}(n_1, n_2)$ by $\text{sim}(a, b)\Delta_{\text{PTK}}(n_1, n_2)$.

1.4 Hungarian Algorithm

When comparing many suspicious sentences with many source sentences using the methods described above, one ends up with a large number of scores. From these scores one must also still decide which of the source sentences a suspicious sentence (probably) plagiarizes. In other words, one needs to chose suspicious/source pairs such that:

1. the most similar (highest scoring) sentences are paired, under the constraint that

2. each sentence appears only once in the resulting set.

This description is a perfect fit for the classical assignment problem.

One method for solving the assignment problem was introduced nearly sixty years ago [19]. The author, Harold Kuhn, called it the Hungarian method, as it

was based on work by two Hungarian mathematicians [20, 21]. It takes an $n \times n$ matrix of costs and tries to find n entries (i.e., assign a row to a column) such that:

1. no two of these are in the same row and no two are in the same column,
2. these entries are optimal in the sense that their sum is the minimum possible for the matrix in question, given the row/column constraint.

How it does this can be most easily seen from a brief description of the algorithm, followed by an example.
The Hungarian algorithm can be divided into five steps:

1. Find the smallest value in each row and subtract it from that row.
2. Find the smallest value in each column and subtract it from that column.
3. Cover all the zeroes produced in steps 1 and 2, by drawing lines over the row or column in which a zero appears, using the minimum number of lines possible.
4. If the number of lines used is equal to n then an optimal assignment is possible and the algorithm stops, else it proceeds to step 5.
5. Find the smallest value not covered by any line. Subtract this value from all uncovered rows and add it to all covered columns. Return to step 3.

Example 1.3. As an example, let us consider the very simple case of deciding which of three suspicious sentences (t_1, t_2 and t_3) plagiarizes which of three source sentences (s_1, s_2 and s_3). Representing the (made up) similarity scores as a matrix gives one:

	s_1	s_2	s_3
t_1	0.45	0.75	0.3
t_2	0.6	0.5	0.9
t_3	0.05	0.1	0.8

To apply the Hungarian algorithm, one needs to convert the above to a cost matrix by subtracting all the entries from 1 (since 1 is the maximum value a similarity score can have):

	s_1	s_2	s_3
t_1	0.55	0.25	0.7
t_2	0.4	0.5	0.1
t_3	0.95	0.9	0.2

With the cost matrix in hand, one can start to apply the steps of the Hungarian algorithm.

1. **Step 1** - subtract the smallest value in each row from that row:

	s_1	s_2	s_3
t_1	0.55	0.25	0.7
t_2	0.4	0.5	0.1
t_3	0.95	0.9	0.2

$\rightarrow$

	s_1	s_2	s_3
t_1	0.3	0.0	0.45
t_2	0.3	0.4	0.0
t_3	0.75	0.7	0.0

2. **Step 2** - subtract the smallest value in each column from that column:

	s_1	s_2	s_3
t_1	0.3	0.0	0.45
t_2	0.3	0.4	0.0
t_3	0.75	0.7	0.0

$\rightarrow$

	s_1	s_2	s_3
t_1	0.0	0.0	0.45
t_2	0.0	0.4	0.0
t_3	0.45	0.7	0.0

3. **Step 3** - cover the zeroes using minimal lines:

	s_1	s_2	s_3
t_1	0.0	0.0	0.45
t_2	0.0	0.4	0.0
t_3	0.45	0.7	0.0

4. **Step 4** - since the minimum number of lines needed equals the size of the matrix (3), an optimal assignment is possible and the algorithm stops.

5. **Optimal assignment** - using the above matrix (more detail below) and the fact that there is a one-to-one correspondence between costs and similarities,

the optimal assignment of sentence pairs is:

	s_1	s_2	s_3
t_1	0.55	0.25	0.7
t_2	0.4	0.5	0.1
t_3	0.95	0.9	0.2

$\rightarrow$

	s_1	s_2	s_3
t_1	0.45	0.75	0.3
t_2	0.6	0.5	0.9
t_3	0.05	0.1	0.8

In other words, we end up with matched sentence pairs (t_1, s_2), (t_2, s_1) and (t_3, s_3).

Going from the step where one has the correct number of covering lines (i.e., an optimal assignment is possible) to the actual optimal assignment requires some more explanation. Any entry which is zero could in principle become part of the optimal assignment. However, only one zero in each row/column can be used. In the example above for instance, if one uses the zero at entry (t_1, s_1), then one cannot use the entry at (t_1, s_2). This would mean not having an assignment for column two at all (since there are no other zeroes in that column), which is not allowed. The algorithm therefore selects the row/column with the fewest choices, picks an entry, updates the choices and repeats this process until all assignments have been made.

In this simple example, there was no need to go to step 5 of the algorithm. Also, there was also only one optimal assignment. For different and/or larger matrices, neither of these need to hold. However, the Hungarian algorithm always finds *an* optimal assignment, even if it is not unique for a given matrix.

The above example used a square matrix which equated to an equal number of suspicious and source sentences. This will not be the case in general. However, one can always pad the matrix with rows or columns consisting of only some fixed cost greater than any actual cost to make the matrix square. In such a case – after the algorithm terminates – one then only uses assignments that fall inside the original range of rows and columns.

The running time of the Hungarian algorithm is $O(n^3)$, where n is the larger of the number of rows or the number of columns. This means that it is quite slow and thus efforts are made to keep the matrices as small as possible whenever the Hungarian algorithm is employed.

1.5 PAN Corpora Details

The efficacy of the implemented plagiarism detection system will be evaluated using the corpora created for the PAN competitions – specifically the competitions from 2009 and 2011 [1, 2, 3]. In these corpora, text documents are divided into *suspicious* and *source* groups. Both groups start as documents taken from Project Gutenberg [22]. The source group is not modified, but the suspicious documents may contain plagiarism from one or more of the documents in the source group.

The suspicious documents contain plagiarism with various levels of obfuscation, namely 'none', 'low' and 'high'. The exact meaning of these labels is unfortunately not clear from any of the competition overviews [1, 2, 3]. The amount of plagiarism in a suspicious document ranges from 0% to 100% and each plagiarism instance spans between 50 and 5000 words.

Plagiarism in the corpora is mostly automatically generated and is constructed using a combination of the following three techniques [1]:

1. **Random Text Operations**: A plagiarism instance is created by taking a source passage and shuffling, inserting, removing or replacing words or short phrases at random. Insertions and replacements might be taken from the document into which the plagiarism instance is placed.

2. **Semantic Word Variation**: A plagiarism instance is created by taking a source passage and replacing each word with a randomly chosen synonym, hypernym, antonym or hyponym. If none of these can be found the word remains unchanged.

3. **PoS-preserving Word Shuffling** Given a source passage and its sequence of part-of-speech (PoS) tags, a plagiarism instance is created by shuffling the words in this passage while retaining the original PoS sequence.

There are also two types of plagiarism which do not clearly fall into a specific obfuscation category. The first of these is so-called 'translation' plagiarism. The PAN corpora contain a small amount of non-English text (German, Spanish and French). A 'translation' plagiarism is inserted into a suspicious file by directly translating a piece of text (into one of the other languages) from a source file.

In the PAN 2009 corpus, these 'translation' plagiarism instances are treated as belonging to the 'none' obfuscation category. For PAN 2011 on the other hand, such a translation may or may not also be manually obfuscated. It is also no longer part of the 'none' category anymore, but in its own 'translation' category.

The second is called 'simulated' plagiarism. This is plagiarism which was crafted by hand from a given source file, specifically for the PAN competition. Simulated plagiarism is not present in PAN 2009 and was introduced for the PAN 2011 competition.

Although the ways in which plagiarism is automatically generated – as described above – do not exactly match the way a human would do it overall, they do consist of techniques a human might use, albeit randomly instead of with forethought. They also allow for the generation of vast amounts of plagiarism required for corpora of the size used by the PAN competitions.

The PAN 2009 competition consisted of 7214 suspicious documents and 7215 source documents. Between the three obfuscation categories, 43% are labeled as 'none', 38% as 'low' and 19% as high. Roughly 5 of the 43 percentage points in the 'none' category are actually 'translation' plagiarism.

The PAN 2011 competition was made significantly more difficult, by greatly reducing the number of 'none' instances in favour of 'high' instances as well as the addition of 'simulated' instances. Furthermore, the 2011 competition is also approximately 50% larger overall, with 11093 suspicious and 11093 source documents. The distribution of plagiarism into the categories, 'none', 'low', 'high', 'translation' and 'simulated' is roughly 2%, 40%, 38%, 10% an 10% respectively.

A quick summary of the distribution of plagiarism into the various categories for easy comparison can be seen in Table 1.1.

	None	Low	High	Translation	Simulated
PAN 2009	43%	38%	19%	5% (part of None)	n/a
PAN 2011	2%	40%	38%	10%	10%

Table 1.1: Distribution of plagiarism into categories for PAN 2009 and 2011.

1.5.1 Measures of Detection Quality

In order to make the results obtained from the implemented system and the results obtained from the entries to the PAN competitions directly comparable, the same measures for the quality of detection are used.

The competitions define five measures in total [1], namely *micro/macro-averaged* recall/precision and the so-called *granularity*. There is also a score based on these called the *pladget* score, defined below in equation (1.42).

Micro-averaged recall and precision follow intuitively from their usual definitions when viewing a piece of text as a sequence of characters and a detection/plagiarism instance as a subset of these characters. Consider the general definitions of precision and recall,

$$\text{precision} \equiv \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{1.25}$$

$$\text{recall} \equiv \frac{\text{TP}}{\text{TP} + \text{FN}}\,, \tag{1.26}$$

where TP, FP, TN and FN are the number of true positives, false positives, true negatives and false negatives respectively. In the current context, the definitions for true/false positives/negatives are as follows. True positives are the characters that belong to a plagiarism instance that are also in a detection. False positives are characters that are in a detection, but are not plagiarism characters. True negatives are characters that are neither plagiarism nor detected. False negatives are plagiarism characters that are not detected.

Given these definitions, the *micro-averaged* precision and recall are then,

$$\mu\text{-precision} \equiv \frac{\text{total TP-chars}}{\text{total TP-chars} + \text{total FP-chars}} \tag{1.27}$$

$$\mu\text{-recall} \equiv \frac{\text{total TP-chars}}{\text{total TP-chars} + \text{total FN-chars}}, \tag{1.28}$$

where the 'total' refers to the fact that one sums the number of relevant (TP-chars for total TP-chars, etc.) characters coming from all detections and plagiarism instances while ignoring duplicates in the detections.

Macro-averaged recall and precision have slightly more complicated definitions.

If one defines the set of plagiarism instances as S and the set of detection instances as R, then the *macro-averaged* recall is given by,

$$m\text{-recall}(S,R) \equiv \frac{1}{|S|}\sum_{s\in S}\frac{s \text{ positional overlap with } \bigcup_{r\in R} r}{|s|}\,. \tag{1.29}$$

In other words, *macro-averaged* recall is the sum of the fractions of detected plagiarism per case, averaged over the number of plagiarism instances. This means that, when using *macro-averaged* measures, all plagiarism instances are considered equal, irrespective of length.

Macro-averaged precision does not follow as straightforwardly as the recall does. This is due to the fact that a set of detections will typically not have one unique detection per plagiarism case. If one swaps the roles of S and R, however, one can define the precision as the recall of R under S, since plagiarism instances (at least in these corpora) are defined to be unique and non-overlapping. Mathematically therefore,

$$m\text{-precision}(S,R) \equiv m\text{-recall}(R,S) = \frac{1}{|R|}\sum_{r\in R}\frac{r \text{ positional overlap with } \bigcup_{s\in S} s}{|r|}\,. \tag{1.30}$$

While this is not the usual relationship between recall and precision, this make sense as a precision measure, since extraneous detections will not be 'recalled' by the actual plagiarism instances, resulting in lower precision.

The final measure that the competitions define is called the *granularity*. This measure is designed to take into account the fact that neither the *micro-* or *macro-averaged* view take into account the number of times a plagiarism instance may be detected. Consider a detection $r \in R$ and a plagiarism instance $s \in S$, then a cover[1] of s, C_s, is defined as the set of all r that overlap with s. *Granularity* is then a measure of the average size of these covers. More rigorously, let $S_R \subseteq S$ be the set of plagiarism instances which have at least one overlapping detection. The

[1]Not strictly a cover in the mathematical sense, but rather a collection of sets (the r's) that all intersect with a specific set (the s).

granularity is then,

$$\text{granularity}(S, R) \equiv \frac{1}{|S_R|} \sum_{s \in S_R} |C_s| \,. \tag{1.31}$$

Example 1.4. To aid in the understanding of equations (1.29) to (1.31), consider the following example. Let the notation $[a, b]$ indicate a contiguous sequence of characters starting at offset a (from the start of some text file) and ending at offset b, where the characters at a and b are both included. Suppose one has two plagiarism instances: one denoted by $s_1 = [100, 250]$ and another denoted by $s_2 = [400, 600]$. Furthermore, suppose there are three detections r_i, where $r_1 = [80, 150]$, $r_2 = [140, 200]$ and $r_3 = [300, 500]$. The size of s_1 is therefore $|s_1| = 151$ and similarly $|s_2| = 201$, $|r_1| = 71$, $|r_2| = 61$ and $|r_3| = 201$.

To calculate the *macro-averaged* recall, one first determines the set of characters that match those found in detections for each plagiarism instance. For s_1 this is $[100, 150]$ from r_1 and the entire r_2 with $[140, 150]$ being duplicates. Let us denote the characters from s_1 that overlap with detections by $o_1 = [100, 200]$. One can do the same for s_2 to arrive at $o_2 = [400, 500]$. The number of detected characters for both s_1 and s_2 are therefore $|o_1| = |o_2| = 101$. The *macro-averaged* recall is therefore,

$$m\text{-recall}(\{s_1, s_2\}, \{r_1, r_2, r_3\}) = \frac{1}{|\{s_1, s_2\}|} \left(\frac{|o_1|}{|s_1|} + \frac{|o_2|}{|s_2|} \right) \tag{1.32}$$

$$= \frac{1}{2} \left(\frac{101}{151} + \frac{101}{201} \right) \tag{1.33}$$

$$= 0.5857 \,. \tag{1.34}$$

For *macro-averaged* precision, one reverses the roles of the plagiarism instances and the detections. In a similar fashion as before, one finds the set of overlapping characters as $o_3 = [100, 150]$ for r_1, $o_4 = [140, 200]$ for r_2 and $o_5 = [400, 500]$ for r_3.

The *macro-averaged* precision is therefore,

$$m\text{-precision}(\{s_1, s_2\}, \{r_1, r_2, r_3\}) = \frac{1}{|\{r_1, r_2, r_3\}|}\left(\frac{|o_3|}{|r_1|} + \frac{|o_4|}{|r_2|} + \frac{|o_5|}{|r_3|}\right) \tag{1.35}$$

$$= \frac{1}{3}\left(\frac{51}{71} + \frac{61}{61} + \frac{101}{201}\right) \tag{1.36}$$

$$= 0.7403\,. \tag{1.37}$$

To find the granularity of the set of detections in this example, one needs to count the number of plagiarism instances that overlap with at least one detection. This is the case for both s_1 and s_2 and therefore $S_R = \{s_1, s_2\}$. Next one needs to find how many detections overlap with each plagiarism instance. For s_1, both r_1 and r_2 overlap with it. For s_2 only r_3 has any overlap. This means that $C_{s_1} = \{r_1, r_2\}$ and $C_{s_2} = \{r_3\}$. Putting everything together gives,

$$\text{granularity}(\{s_1, s_2\}, \{r_1, r_2, r_3\}) = \frac{1}{|\{s_1, s_2\}|}\left(|C_{s_1}| + |C_{s_2}|\right) \tag{1.38}$$

$$= \frac{1}{2}(2 + 1) \tag{1.39}$$

$$= 1.5 \tag{1.40}$$

For comparisons between results, we shall use two different scores. The first is an F-score based on the *macro-averaged* recall and precision:

$$F_\beta \equiv (1 + \beta^2)\frac{m - \text{precision} \cdot m - \text{recall}}{\beta^2 \cdot m - \text{precision} + m - \text{recall}}\,. \tag{1.41}$$

In particular, we use the F_1 score.

The second score is what is known as the plagdet score and was designed by the PAN organizers in order to take the granularity of detections into account. As such, it is given by

$$\text{plagdet} \equiv \frac{F_1}{\log_2(1 + \text{granularity})}\,. \tag{1.42}$$

When a granularity of 1 is achieved, this reverts to the F_1 score.

1.6 Related Work

Plagiarism detection techniques can usually be divided into two categories. The first consists of methods that aim to compare entire documents and include algorithms such as hashing or fingerprinting [23, 24]. The second category of methods is usually applied after the first has produced a candidate set of potential source documents for a given suspicious document. These methods typically involve a more detailed textual analysis of a suspicious document in comparison with a source. N-grams and string similarity metrics are popular choices here [24, 25].

A more detailed description of these techniques are given below in the context of the PAN competitions.

1.6.1 PAN Competitions

Since the PAN competitions' corpora make up the bulk of the data on which the plagiarism detector was trained and tested, the work of the participants in these competitions is of particular note.

1.6.1.1 PAN 2009

The winning entry [26] for the PAN 2009 competition relied on character 16-grams at each stage in the detection process. In the retrieval stage, an exhaustive comparison of suspicious documents to source documents is made, yielding a large document pair similarity matrix. This similarity measure used simply counts the number of shared 16-grams present in each document pair. From this matrix, the contestants rank the suspicious documents for a given source in decreasing order of similarity and choose the top 51 documents for more detailed analysis. This analysis starts by finding the locations (character offsets) in each document where an exact 16-gram match occurs. Specifically, a list of location pairs is created by outputting the location of the first instance of a 16-gram in one document with the location of the first match in the other, then the location of the second instance with the second match and so on. Merging lists of this kind into larger groups to flag as plagiarism is done by finding groups of nearly contiguous location pairs. The largest group is found via a Monte Carlo optimization and removed from the

list. This process repeats until no more groups can be found, based on some size and contiguity heuristics.

The runner up [27] in the PAN 2009 competition used word 5-grams instead of character 16-grams, which these contestants call chunks. The chunks are enriched with information about where in a document they appear. Chunks are hashed, and an index from the document ID to the hash is created. An inverse index – from hash to document ID – is also constructed. To retrieve a pair of similar documents, the lists of hashes are compared and documents with more than a certain number of common chunks (hashes) are labeled as similar. The contestants chose 20 common chunks as their cutoff value and document pairs meeting this criterion are analyzed further. When examining a document pair in more detail, the inverse index is used to find the source document and list of chunks that are similar. Sections of text that constitute plagiarism are then computed by matching 'dense enough' intervals of chunks in one document with 'dense enough' intervals in the other. The authors chose an interval of chunks that have no more than 49 missing chunks between any subsequent members of the interval as 'dense enough'. If there are overlapping intervals, then only the largest is kept.

1.6.1.2 PAN 2010

The contestants that won PAN 2010 were the runners up in PAN 2009. As such, their detection method is mostly the same with a few small changes [28]. When considering overlapping detections, the detector previously kept the larger of the two. In the updated version, both detections are discarded if they are shorter than 600 characters long. Furthermore, the authors changed when and how detections were merged. In their 2010 detector, detections were merged if the gap between two was less than 600 characters. Detections were also merged if the gap was less than 4000 characters and the average length of the two adjacent detections was more than twice this size.

For PAN 2010, the second place entry was submitted by Zou et al. [29]. In order to detect similar documents, overlapping word 5-grams are found for each document and hashed – forming a set of fingerprints [30]. Documents are labeled as similar when there are not too many differing fingerprints. The character offset

locations of the 5-grams are also stored for each document. During the detailed analysis of a pair of documents, the locations of matching 5-grams are used to find clusters of detections that satisfy certain criteria. The detections inside a cluster are merged to form a single detection.

1.6.1.3 PAN 2011

The detector that won the PAN 2011 competition [31] used single words (i.e., word uni-grams) as their basic unit of detection. To facilitate this, all words in the corpus are first stemmed and then 'synonym normalization' is performed. The authors do not explicitly mention how document pairs are chosen for greater scrutiny. Documents are divided into passages of a constant number of words and a pair of passages are flagged when they share a number of words larger than a chosen threshold. Passages belonging to a flagged pair are divided into sub-passages that start or end on one of the shared words. If these sub-passages correspond to a large enough fraction of the full passage against which it is matched (i.e., a suspicious sub-passage must be larger than a certain fraction of the full source passage with which it is paired and vice versa) and as long as these sub-passages pairs still contain more than a certain number of shared words (the authors chose 15), then the sub-passage pair is flagged as plagiarism. Post-processing involves merging adjacent passages and removal of overlapping ones.

The entry coming second in the PAN 2011 competition [32] was an updated version of the detector that won the PAN 2009 competition. This updated version uses a different document similarity measure. Whereas before, the number of shared character 16-grams was used, this version determines the number of moving windows (of 256 characters) that contain at least 64 shared 16-grams. Source documents are then ranked by decreasing similarity for every suspicious document and vice versa. Documents are compared in more detail if either ranking falls within the top n, where n is assumed to once again be 51. The detailed analysis and detection merging parts of their detector remained mostly unchanged.

1.7 Summary

This chapter provided details regarding the techniques that form the building blocks for the methods the detector uses to find similar sentences. These methods were split into two categories, namely those based on feature vectors and those based on tree kernels. The vector-based methods rely on either Word2Vec or Doc2Vec and a board overview of each was given.

The tree kernel-based methods operate on the parse trees of sentences. As such, two kinds of parse trees, namely constituency and dependency trees, were described, followed by sections on how the tree kernels themselves work.

While the detector is running, it compares many suspicious sentences with many potential source sentences and computes similarity scores for every pair. In order to find the optimal set of pairs (based on the similarity scores), it employs the Hungarian algorithm. This algorithm was described in its own section, which also contained a small worked example.

The detector was evaluated against the PAN 2009 and 2011 corpora. These corpora contain various kinds of automatically generated as well as hand-crafted plagiarism instances and a breakdown of the distribution of obfuscation categories was reported. The methods by which automatically generated instances are constructed were also described. The PAN corpora define a number of measures useful for quantifying the performance of a plagiarism detector and the details of these measures were provided.

This chapter was concluded by giving a short discussion on related work, focusing mainly on the detection methods used by the competitors in PAN 2009 to 2011.

Chapter 2

Methodology

This chapter details how the developed plagiarism detector performs its task, beginning with a collection of suspicious and (potential) source documents until finally producing a collection of annotations listing the (potential) plagiarism instances.

These documents could come from a pre-compiled corpus, such as is the case with the PAN competitions. Alternatively, the suspicious documents might be student submissions for university work to be checked against other students' submissions. Another possibility could be articles submitted to a publisher to be compared with a database that the publisher keeps.

The detection process is broken up into four broad stages: pre-processing, information retrieval (IR), the main plagiarism detection step and post-processing. A diagram outlining the entire detection process can be seen in Figure 2.1.

The Doc2Vec method used by the IR phase is only really suitable in the case where one has an offline corpus. In a real-world setting one would, therefore, either have to use an existing/build one's own corpus or use a different method entirely.

Most of the tasks before the main plagiarism detection step are performed using Python. Some of the larger libraries used include NumPy (v.1.11.1) [33], gensim (v.0.12.4) [11] (for Word2Vec and Doc2Vec training) and NLTK (v.3.2.1) [34] (for most of the corpus file handling and text processing).

For the plagiarism detection itself, Java is used. This includes code written specifically for the detector, as well as a number of libraries. Two of the major libraries used are KeLP (v.2.0.0) [35] (for the tree kernel calculations) and ND4J

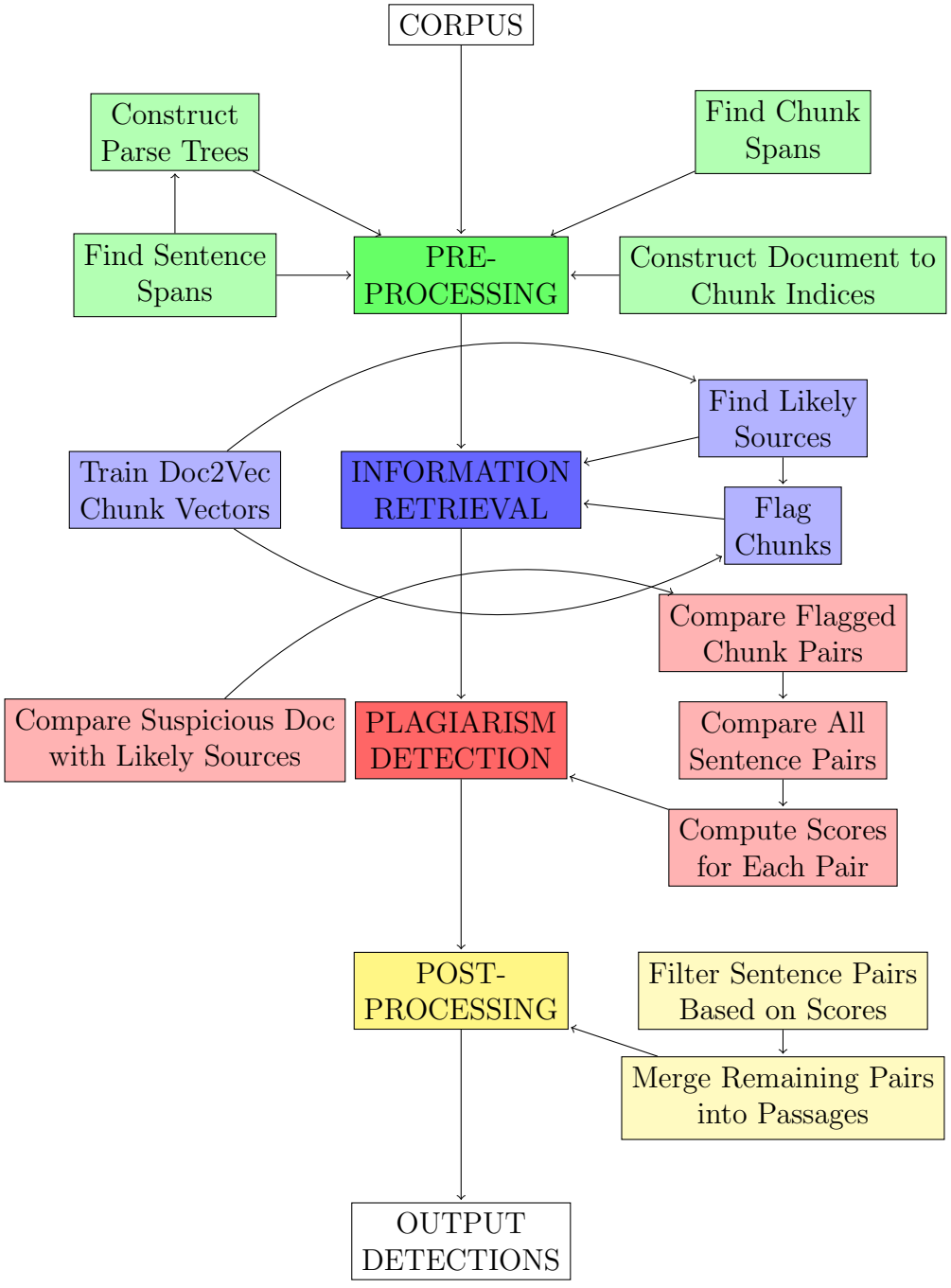


Figure 2.1: Overview diagram of the four detector stages. Each stage has its own section, explaining it in detail.

(v.0.4-rc3.8) [36] (for everything vector related). While strictly speaking part of the pre-processing, the parsing of text was done using the parsers included in the Stanford CoreNLP (v.3.6.0) library [37], which is also written in Java.

2.1 Preprocessing

For this thesis, the assumption is that one has a collection of source text files and a collection of so-called suspicious text files – a plagiarism corpus in other words. One then wants to find passages (contiguous sets of characters) in the suspicious files that plagiarize passages in the source files. In order to facilitate subsequent steps in the detection process, various pre-processing tasks are performed.

Since the plagiarism detector performs its detections at the sentence level, one of the first tasks is to output (to file) the sentence character spans for each document in the corpus. The IR step (see Section 2.2.1) ultimately operates on chunks of words (a larger number of consecutive words – typically 50). However, these chunks may (and most often do) cross sentence boundaries. The chunk character spans – which are also output to file – together with the sentence spans allows one to find the sentences which cover a certain chunk. Both the sentence spans and chunk spans are obtained using tokenizers of the NLTK library.

Another important place where sentence spans are used, is during the parsing of text into parse trees. The neural network dependency parser [38] of CoreNLP produces the dependency trees used later on during plagiarism detection (Section 2.3). Tokenization (dividing text up into its constituent words and symbols) is not standardized, and therefore text is split up slightly differently by the tokenizers in NLTK and those in CoreNLP (as part of the parsing process). CoreNLP is therefore instructed not to detect sentence boundaries, but instead to treat the characters specified by any given span (as found in the sentence span files) as one complete sentence.

At this point, some intricacies regarding the character encoding of the PAN corpora and how these influence all the above pre-processing should be pointed out. NLTK provides a number of built-in ways to read various parts (e.g., words, sentences, etc.) of documents contained in a corpus. However, these methods will report character errors for some files when the encoding that the PAN corpora

apparently use (UTF-8 with a byte-order-mark) is specified. Ignoring these errors is undesirable, since this can change character offsets relative to the start of a file. This, in turn, is important, because the PAN corpora specify plagiarism instances as contiguous spans of these character offsets. Making systematic errors in these offsets can, therefore, lead to a decrease in both recall and precision when comparing detections to the ground truth. The workaround here – which avoids character errors entirely – is to find the words or sentences required by directly using the tokenizers provided by NLTK.

The Doc2Vec vectors produced for each chunk (see Section 2.2.1) are stored in a large matrix, where each row is a vector for a specific chunk, in the order that these chunks are passed to Doc2Vec during training. In order to facilitate the efficient comparison between the vectors from chunks in specific files, a dictionary is built in the pre-processing stage that maps a file name to a tuple containing the index of the first chunk for that file as well as the total number of chunks in the file.

2.2 Information Retrieval

The information retrieval (IR) step is there to perform a broad sweep over the given corpus to find the areas that are most likely to be plagiarized. More specifically, this step aims to remove as much of the corpus as possible, while keeping those parts that are likely plagiarism. Included in this removal are parts of both suspicious and source documents. This reduction in the amount of processing the main plagiarism detection step needs to do is crucial for the scalability of the system. The PAN 2009 corpus, for example, has 7214 suspicious and 7215 source files, resulting in approximately 150 000 000 000 000 sentence comparisons. For this many comparisons, an exhaustive comparison would take approximately 47.5 years if one assumes 10 μs per comparison.

The main problem here is, of course, the $O(n^2)$ nature of comparing every sentence in suspicious documents to every sentence in source documents (assuming roughly n sentences overall in each document group). The IR step does not change this time complexity, but rather tries to make n as small as possible.

2.2.1 Doc2Vec Vectors

To whittle a corpus down to a manageable size, Doc2Vec (see Section 1.1.2) is used. Doc2Vec vectors are found for all consecutive chunks (a number of consecutive words, chosen to be 50) appearing in the corpus in two steps.

First, using the `gensim` library, normal Doc2Vec training is performed on the chunks of the source documents only. All stopwords (words which carry no, or very little, semantic information) are removed from text before being broken up into chunks. A list of these stopwords are provided by NLTK.

In what follows, some of the parameter choices made for Doc2Vec are listed. However, due to the length of time a full detection run takes (in the order of a week), it is not possible to investigate comprehensively the effects of changing these parameters. The values used were found by performing trials on small portions of the PAN 2009 training corpus.

Before the training starts in earnest, Doc2Vec performs a single pass over all the text in the corpus and builds a vocabulary and the number of times each word appears. One can specify a cutoff value for this occurrence number, and words occurring fewer times than this threshold will be ignored by the model being trained. In most of the Doc2Vec models trained for the various corpora, this cutoff was set to 40. This leaves around 97% of the total number of words in the PAN corpora, while reducing the vocabulary size (unique words visible to Doc2Vec) from around 1.5 million to around 100 thousand. The argument for doing this is twofold: it reduces the model size and words with low occurrence counts do not typically have enough examples to produce accurate feature vectors.

Another important parameter is the number of epochs (full passes) for which training is done. For the PAN 2009 corpus, 20 performed well. For the PAN 2011 corpus on the other hand, due to its larger size, 40 epochs was used. Unfortunately, the `gensim` library does not provide access (at the time of writing) to any kind of metric that could be used to evaluate training performance. The number of epochs does therefore not have any kind of rigorous explanation for why a certain choice was made.

For the dimension of the vectors, a number was chosen that was as large as possible while still allowing the entire set fit into memory (16GB on the system

used for development of the detector). For PAN 2009 this was 800 and for PAN 2011 (since it is larger) this was 600.

Perhaps the most crucial choice is that of the architecture (see Section 1.1.2) – Distributed Memory (DM) or Distributed Bag-of-Words (DBOW). A handful of tests on a subset of the PAN 2009 training corpus showed that the DBOW architecture vastly outperformed DM given that other training parameters remained the same. The DBOW architecture is therefore used for all Doc2Vec models mentioned throughout this thesis. The hypothesis here is that the range of topics contained in the PAN corpora is diverse and the DBOW architecture helps by allowing more examples per chunk to be considered.

The end result of Doc2Vec training is a large matrix with one vector (row) for each chunk.

At the start of this section, it was stated that normal Doc2Vec training is done for chunks from source documents only. That is because the vectors for the suspicious documents' chunks are obtained slightly differently. Given a trained Doc2Vec model, one can use its built-in `infer_vector()` method to produce a vector given a list of words. This method essentially takes the list and performs a specified number of training steps (epochs) with it as input. The method does not alter anything in the original model and just returns the resulting vector. Passing the chunks from the suspicious documents one-by-one to `infer_vector()` and saving the output gives a set of vectors that one can use for comparison with those from the source documents. This difference in training seems reasonable, given the likely real-world use case where one would have an established source corpus and would want to check submitted documents against it for plagiarism.

2.2.1.1 Finding Likely Sources

With a set of suspicious/source chunk vectors and the dictionaries that map a document name to a chunk vector index range, one can compare all the chunks in one document with those of another. Consider the case where a document, say `suspicious-document12345.txt` has 207 chunks. The entry for this document in the dictionary could be something such as $(1034902, 207)$, which means that in the (suspicious) chunk vector matrix, rows 1034902 to 1035108 are the

vectors for `suspicious-document12345.txt` specifically. If one wants to compare `suspicious-document12345.txt` with `source-document12121.txt`, with say entry $(4101995, 1309)$, then one needs to take the cosine similarity of all $207 \times 1309 = 270963$ pairs.

By normalizing both sets (suspicious and source) of chunk vectors, calculating all the similarities can be performed in a single matrix multiplication. To see this, one simply needs to remember the definition of matrix multiplication. If matrix A is size $n \times k$ and matrix B is size $k \times m$, then the (i, j)th entry in the product of the two matrices is

$$(AB)_{ij} \equiv \sum_{h=1}^{k} a_{ih} b_{hj}. \tag{2.1}$$

Looking back at (1.3), it is clear that this amounts to the inner product of matrix A's row i, a_{ih}, with matrix B's column j, b_{hj}. If every chunk vector is size, say k, then taking A as the $207 \times k$ matrix of suspicious chunk vectors and B as the $k \times 1309$ matrix of the transpose of the source chunk vectors gives the result one wants. This is typically much faster than performing the inner products one by one, since libraries such as NumPy use special routines for matrix operations that parallelize much of the computation.

By performing the kind of calculation described above for every suspicious/source document combination, one can rank the potential sources for every suspicious document based on the maximum similarity found in each calculation. Afterwards, for every suspicious document a file is created containing a list of the top 100 sources.

2.2.1.2 Flagging Chunks

Flagging chunks for more in-depth scrutiny (as per Section 2.3), is very similar to finding the likely sources (Section 2.2.1.1). Just as before, all the cosine similarities between the chunks from a suspicious document and a source document are computed (i.e., recalculated) in exactly the same way. The only difference now is that it is no longer just the maximum of these similarities that is important. Instead, one identifies all pairs of indices yielding a similarity above a specified threshold.

If the threshold is for example 0.3, and the entry in the 5th row, 19th column is

0.419, then $(5, 19)$ becomes a flagged chunk pair for the current suspicious/source document combination. The output from this step is a file for every document combination that corresponds to a suspicious file and one of its top n (n is usually chosen as 20) ranked source documents (see Section 2.2.1.1). These files contain multiple lines with two numbers each – the flagged chunk indices.

2.3 Plagiarism Detection

At this stage in the detection process as a whole, an exhaustive comparison is made of all remaining sentence pairs, as specified by the flagged chunks from the previous (IR) stage (see Section 2.2.1.2). To clarify this a bit, consider the following example where there are only two files in some hypothetical corpus – one suspicious and one source file. Furthermore, let us suppose that for $n = 0, 1, \ldots$, chunks in the suspicious and source files are labeled by c_n and d_n respectively, while the sentences in the suspicious and source files are t_n and s_n respectively. From the IR step, one might have a list of flagged chunks such as the following,

```
0   193
1   3
1   5
1   57
16  20
17  13
17  20
etc.
```

where the first column indicates the n'th chunk for the suspicious file and the second column likewise for the source file. If chunk c_1 consist of sentences t_6 to t_9 and d_3 consists of sentences s_{17} to s_{22} then – due to the `'1 3'` entry in the above list – all sentence pairs $\{(t_i, s_j) \,|\, 6 \leq i \leq 9;\ 17 \leq j \leq 22\}$ will need to be compared. It is important to note that sentences are only compared when they belong to a flagged chunk *pair*. For example, even though c_{16} and d_5 are in the above list of flagged chunks, since they are not flagged together as a pair, sentence pairs from this combination of chunks are never compared.

The above depicts the case for one suspicious file and one source file. Typically, for any suspicious file T the system would consider all source files S_i, where S_i is one of the top n likely sources (see Section 2.2.1.1) The detector will then compare the relevant sentence pairs identified by all the flagged chunks between T and S_i (for each i). A full run of the detector therefore compares all the flagged chunks pairs of each suspicious file and their top n source files. A value of 10 for n was used most of the time, since it gave a good tradeoff between maximum possible recall, increased precision from less false positive detections, and speed.

The previous few paragraphs give a very abstract view of the main plagiarism detection step. The rest of this section concerns itself with the finer details.

2.3.1 Initialization

The plagiarism detection step works exclusively with data generated in the pre-processing (Section 2.1) and IR (Section 2.2) stages. In order to keep things flexible, a `properties` file is used to store meta-data that is required during execution. These include things such as the various input/output directories, classifiers to use, document number range (a [`suspicious`|`source`]`-documentXXXXX` nomenclature is used by the PAN corpora for the different files, where `XXXXX` is the document number) and number of source files to compare a suspicious document with. As the detector iterates over the document number of the suspicious documents, it loads the necessary data from the previous stages into memory.

First among these is the file that specifies the likely sources for the current suspicious document. The detector then also (sub-)iterates over these source documents and loads the needed data pertaining to them.

For a suspicious-source document pair, one vital piece of data is the specific chunks in each to compare. An example of what this data looks like was given at the start of this section (Section 2.3): the index of the chunk in the suspicious document in the first column and the corresponding index of the chunk in the source document in the second column.

Two pieces of data that are always loaded are the current suspicious and source document's chunk and sentence spans (start/end character offsets from start of file). As was alluded to in Section 2.1, but not explained in detail, these two pieces of

information lets one determine which sentences belong to a specific chunk. Consider the following example where a chunk spans characters 650 to 1294. For the same file, the (relevant) sentence spans are as follows:

```
0     212
213   269
270   461
462   655
656   840
841   1080
1082  1240
1241  1395
etc...
```

where the first column indicates the start offset and the second column indicates the end offset. Borrowing from the notation established earlier, it is clear that the sentence spans for s_3 until s_7 (characters 462 to 1395) would cover the chunk in question. Since both columns are sorted, finding the largest start offset smaller than 650 and the smallest end offset larger than 1294 can be found efficiently with a binary search for 650 in the first column and 1294 in the second column.

Next, the data required by the specified classifiers get loaded. A tree kernel based classifier, for example, needs the parse trees of sentences. Similarly, a vector-based classifier might need word vectors to perform its function. Some of this data is needed for the entire run of the detector; such as in the case of word vectors. These word vectors are also shared between classifiers that use them. Other data, such as parse trees, are only stored on a per-sentence basis.

The initialization does not consist solely of loading data from file, but also performs the calculation of norms for classifiers that require them, such as the tree kernel classifiers. As a reminder, the norm for a particular tree t given a tree kernel TK is $\sqrt{\text{TK}(t,t)}$ (see equation (1.4)). Pre-calculating the norms provides a significant reduction in the overall running time of the detector, since it means that a norm needs only be calculated once, instead of for each pairing it might appear in. If there are m sentences (and therefore m trees) per chunk (on average) and n chunk pairs in total, this reduces the number of norms that need to be calculated

from $O(nm^2)$ to $O(nm)$.

Another part of the initialization involves the construction of the vector representations for sentences for the vector-based classifiers. For example, a vector-based classifier might operate on the average of all the vectors for the words that appear in a sentence. Different vector-based classifiers require different representations in general, and one representation for each classifier for each sentence is constructed in this initialization stage.

The data loaded or constructed during this stage is discarded when the document to which they pertain is no longer needed. For a suspicious document, this means that the data is kept until all comparisons with its likely sources are concluded. For a source document, the data could in theory be needed again in a comparison with another suspicious document. However, since it is generally not known whether this will occur, the data for a source document is typically discarded immediately after a comparison with a suspicious document.

2.3.2 Sentence Comparisons

Once all the data for a pair of documents is loaded, the comparison process itself is relatively straightforward. The detector goes through the list of chunk index pairs and compares all sentences stored for the current suspicious chunk with those stored for the current source chunk.

What 'compares' means here is that every classifier (details in Section 2.3.3) attached to the detector (as given by the detector meta-data) provides a score for the current sentence pair and an entry is added to a data set for the specific chunk pair. Such an entry contains the suspicious sentence's start and end character offsets, the source sentence's start and end offsets, followed by the scores in the order in which the classifiers are run. An example of what it looks like is the following,

```
257 592 2794 2961 0.2119 0.5471 67 30
257 592 11284 11371 0.1131 0.4906 67 18
257 592 15321 15331 0.1502 0.4517 67 5
257 592 17422 17495 0.1762 0.6892 67 16
257 592 18229 18266 0.1098 0.6118 67 9
```

```
etc.
```

The above example shows a very small part of output produced when comparing `suspicious-document00001.txt` to `source-document01409.txt`, when using the PTK-MF (Section 2.3.3.2) and multiple n-gram vectors (Section 2.3.3.1) classifiers followed by the word number (Section 2.3.3.3) classifier. To clarify, columns 0 to 3 are the sentence offsets, column 4 is the PTK-MF score, column 5 is the multiple n-gram vectors score and columns 6 and 7 are the number of words in the suspicious and source sentence respectively. The first four columns always contain the sentence offset data, but the number of columns that follow can vary depending on the classifiers chosen. If the number of words in the sentences are output, they typically appear as the final columns, but this is not a strict requirement.

When all sentence comparisons for a certain chunk pair are complete, the Hungarian algorithm (Section 1.4) is used to determine the best sentence pairings *for this specific chunk pair*. All other entries generated by the chunk pair are discarded from memory. This means that if a suspicious chunk has, say, five sentences, and the source chunk it was compared with has, say, seven sentences, then $5 \times 7 = 35$ entries will be generated. After running the Hungarian algorithm on this, one is left with five entries (always the smaller of the two numbers).

The Hungarian algorithm takes a matrix of values corresponding to the 'cost' of pairing one sentence with another. It cannot, therefore, use the data entries (such as the ones shown above) as is. The cost for a sentence pair (i.e., a single entry) at this point is defined to be

$$\text{sentence pair cost} \equiv 1 - \frac{1}{|C|} \sum_{i \in C} s_i \,, \tag{2.2}$$

where C is the set of all column indices that contain classifier scores (i.e., not the sentence offset columns (0 to 3) or the word number columns if present), s_i is the score in column i and $|C|$ is the size of the set C.

The remaining entries are used/refined during post-processing (see Section 2.4) to determine the parts of text that become labeled as plagiarism.

2.3.3 Classifiers

The classifiers used by the detector form the core of the detection process. How well the classifiers perform ultimately determines how well plagiarism is detected. All classifiers calculate a score given a pair of sentences. More specifically, they calculate a score given the *representations* of the sentences. For this reason, classifiers can, for the most part, be divided into two categories, namely tree-based and vector-based.

2.3.3.1 Vector-based Classifiers

Word2Vec Classifier During the data initialization stage of the detector (Section 2.3.1) the `createArray` method of this classifier is used to form a vector representation from the words in given sentence. This method takes a list of words and returns the normalized average word vector using a word vector collection (loaded during initialization for all classifiers that use word vectors) to look up vectors for specific words. Unknown words are ignored. Mathematically therefore, if there are k words in a sentence and each word has a vector representation $\mathbf{v}_i$, then the vector representation of the *sentence* is given by,

$$\text{sentence vector} \equiv \mathbf{s} = \frac{1}{k}\sum_{i}^{k}\mathbf{v}_i\,. \tag{2.3}$$

The normalized representation is then,

$$\text{normalized sentence vector} \equiv \mathbf{s}^* = \frac{\mathbf{s}}{\|\mathbf{s}\|}\,. \tag{2.4}$$

In order to be efficient with what data is loaded from file, the list of words is extracted from the parse trees used by the tree-based classifiers, before being passed to the classifier to construct a representation.

When given two representations, i.e. vectors, the classifier calculates the dot product of these and returns the result. As is clear from equation (1.3), the cosine similarity of two vectors corresponds to a simple dot product when the two vectors in question are already normalized. The score that the Word2Vec classifier therefore calculates, is the cosine similarity of the vector representations of two sentences.

Word2Vec with TF-IDF Classifier This classifier calculates its score exactly as the normal Word2Vec one does, however, it constructs a different representation. In addition to a collection of word vectors, this classifier also requires a collection of TF-IDF weights for the words appearing in a corpus.

TF-IDF stands for 'term frequency, inverse document frequency'. This combines the total number of times a term appears in a specific document (term frequency) with a factor based on (the inverse of) the number of documents that term appears in (inverse document frequency). The TF-IDF weight of a word is therefore higher when it appears in a small number of documents, while being lower when it appears only a few times in the document being considered. This reduces the contribution of very common words appearing in virtually all documents and increases it for words appearing many times in only a few documents.

The vector representation that is returned by this classifier is the normalized weighted sum of all word vectors of the words in the given list. The weighted sum is calculated by multiplying a word's TF-IDF weight by the word's vector and adding all of these together. Unknown words are once again ignored. The formula for calculating the TF-IDF weighted representation is very similar to the normal Word2Vec one. Once again, if a sentence has k words, and these words each have vector representation $\mathbf{v}_i$ and TF-IDF weight w_i, then

$$\text{TF-IDF weighted sentence vector} \equiv \mathbf{s}_{\text{TF-IDF}} = \sum_{i}^{k} w_i \mathbf{v}_i \,. \tag{2.5}$$

The normalized form is,

$$\text{normalized TF-IDF weighted sentence vector} \equiv \mathbf{s}^*_{\text{TF-IDF}} = \frac{\mathbf{s}_{\text{TF-IDF}}}{\|\mathbf{s}_{\text{TF-IDF}}\|} \,. \tag{2.6}$$

Given two such representations, the resulting score is therefore the cosine similarity between the TF-IDF weighted average word vectors of the underlying sentences.

Multiple n-gram Vectors Classifier We came upon the idea for the following classifier by considering how to avoid averaging *all* the semantic content in a sentence – as captured by the word vectors – that occurs in the previous two

classifiers.

Unlike the previous two classifiers, this one does not use a single vector to represent a sentence, but many – one vector for each n-gram that covers a sentence. These n-grams are non-overlapping (for computational reasons). Furthermore, they cover a sentence in the same way that sentences cover chunks (see Section 2.3.1). The vector for any given n-gram is the normalized average word vector of all the words in the n-gram. The sentence is therefore represented by a small matrix. If a sentence is covered by m n-grams, for example, the dimensions of the matrix would be $m \times d_{\text{wv}}$, where d_{wv} is the size of the word vectors.

When comparing two sentences using representations such as these, one can obviously no longer use just a single dot product. Consider the case when one sentence has n vectors and another m. The $n \times m$ matrix of cosine similarities of all the pairs of vectors between the two sentences can be found with a single matrix multiplication (same technique used in Section 2.2.1.1), since these vectors are normalized. Assuming, for the sake of argument, that $n < m$, one can find the n best matching (most similar based on cosine similarity) pairs using the Hungarian algorithm (see Section 1.4). Transforming the similarities of these pairs by introducing the cosine distance,

$$\text{cosine distance} \equiv 1 - \text{cosine similarity} \tag{2.7}$$

and using the harmonic mean given by

$$\text{HM}(\{x_i,\ i = 1, 2, \ldots, n\}) \equiv \sum_{i=1}^{n} \frac{n}{x_i}\,, \tag{2.8}$$

one can obtain a score for a sentence pair that emphasizes any similarity that might be present by taking

$$\text{sentence score} = 1 - \text{HM}(\{\text{cosine distance of vector pair } i,\ i = 1, 2, \ldots, n\})\,. \tag{2.9}$$

This is because when a pair has a high similarity, it will have a low distance, and the harmonic mean is biased towards lower values. The geometric mean also shares

this quality (to a lesser degree), however, the harmonic mean is much faster to calculate as it requires no exponentiation.

Vector Classifier Discussion Both the Word2Vec and Word2Vec-with-TF-IDF representations are fairly simple ways of making a sentence embedding from the underlying word embeddings. However, neither works very well for our task. There are many possible reasons for this. Neither pays any heed to word ordering, and as such sentences that might use similar words, but due to their order have different meanings, could falsely receive a high score. One sentence might plagiarize two (or more) sentences, which would lead to a large drop in similarity and subsequently make detection more difficult. In the case of TF-IDF specifically, sentences might appear less similar because a rare word was used in one sentence but not the other.

The multiple n-gram vectors classifier performs much better than the other two classifiers, which use what amounts to a bag-of-words approach. Three different values of n, namely $n = 3, 5, 10$, were tried for the number of words in the n-grams used by this classifier, and $n = 5$ worked the best of these. One could speculate that a window size of five words contains a large enough amount of semantic diversity to keep spurious similarity with other 5-grams low. At the same time, these are few enough words to counteract (to a degree) the 'drowning out' of similarity due to the averaging of the word vectors that make up the n-gram. Using multiple vectors together with the Hungarian algorithm also lets the classifier find good matches between parts of sentences even if a lot of re-ordering has gone into a plagiarism instance. Finally, the use of the harmonic mean to combine the scores from individual n-grams allows for greater retention of high scoring parts of sentences than a normal average does.

2.3.3.2 Tree-Based Classifiers

Partial Tree Kernel Classifier This classifier takes two parse trees, together with the norm of each, as input and produces a score using a partial tree kernel (PTK) (as described in Section 1.3.2). There are three parameters that can be set for the kernel calculation itself, but these are not changeable from outside the detector's code (i.e., via the `properties` file meta-data) at this time. These parameters are the μ and λ mentioned in Section 1.3.2 as well as a parameter which

multiplies the contribution from leaf nodes specifically. The values used in this work are 0.4, 0.4 and 1.0 respectively, which are the default values for the kernel as it is implemented in the KeLP library [35]. Experimentation did not reveal a set of parameters that worked noticeably better.

When a sentence only partially plagiarizes (or fully plagiarizes part of) another sentence, the similarity score calculated by a tree kernel will be lower. This is to be expected, because the sentences are in fact less similar. However, for plagiarism detection, one wants to keep as much of this (partial) similarity as possible. The PTK calculation boils down to a summation of the similarity of all sub-tree pairs between two trees that are rooted at nodes with the same node label.

A slight modification to normal PTK, which we call PTK-MF (partial tree kernel, maximum fragment), tries to emphasize more of the partial similarity that may exist between two trees. For example, when comparing the trees of 'the man walks down the street' and 'the man walks very quickly' using the PTK-MF method, a larger score should be obtained, since PTK-MF will enhance the contribution from the 'the man walks' part of the sentences. Whereas a normal PTK calculation simply sums the similarities (Δ-function contribution) of the sub-tree pairs, a PTK-MF calculation first performs some post-processing on the list of sub-tree pairs with non-zero contribution. The sub-tree pairs in this list are those rooted at nodes with the same node label.

Consider the following scenario. There are two trees, tree A and tree B, with some node c_A in A and some node c_B in B, where the pair (c_A, c_B) is in the list of sub-tree pairs (i.e., they are the roots of the sub-trees). Let a_A be an ancestor node (closer to the root of tree A) of c_A and let a_B be an ancestor node of c_B such that (a_A, a_B) is also in the list of sub-tree pairs. Furthermore, let $|c_A|$ be the number of nodes in the sub-tree rooted at c_A, and similarly $|c_B|$ for c_B and so forth. The post-processing then involves checking whether,

$$\frac{\Delta_{\text{PTK}}(c_A, c_B)}{\sqrt{|c_A||c_B|}} > \frac{\Delta_{\text{PTK}}(a_A, a_B)}{\sqrt{|a_A||a_B|}}. \tag{2.10}$$

If both $|c_A|$ and $|c_B|$ are greater than 5 and if the above inequality holds, then $\Delta_{\text{PTK}}(a_A, a_B)$ is replaced by $\Delta_{\text{PTK}}(c_A, c_B)\frac{\sqrt{|a_A||a_B|}}{\sqrt{|c_A||c_B|}}$ in the list of sub-tree pair con-

tributions. Essentially, this makes the larger ancestor sub-tree pair as similar as its child sub-tree pair, while adjusting for the increase in tree sizes.

Smoothed Partial Tree Kernel Classifier The smoothed partial tree kernel (SPTK) classifier operates exactly as the PTK classifier does, with the difference being that a SPTK is used instead of a PTK. The SPTK implementation in KeLP [35] has one additional parameter compared to PTK, which is the node similarity cutoff.

When comparing two nodes with different node labels, their contribution is multiplied by zero in a PTK. For the SPTK on the other hand, this multiplicative factor is instead determined by an arbitrary similarity function. The PTK can therefore be seen as the special case of the SPTK where this function returns 0 if the node labels are different and 1 if they are the same. The node similarity cutoff parameter specifies the value below which nodes will be treated as if they have a similarity of 0. The SPTK classifier uses the same parameter settings as the PTK classifier (namely, 0.4, 0.4 and 1.0) for the parameters they share. The node similarity cutoff is chosen as 0.1. An important motivation for having such a cutoff is computational – without it the running time of SPTK would become very large.

The similarity function used was the cosine similarity between the word vectors of the respective node labels (i.e., the words in the sentence that the parse tree is generated from).

SPTK is much slower than PTK (between 50 and 100 times), due to having to iterate over many more sub-trees pairs than PTK does as well as having to calculate word pair similarity values each time. The calculation of the the word pair similarities can be sped up by placing the (normalized) word vectors for each word in a sentence in a matrix. The similarities between the words of two sentences can then be found by a single matrix multiplication (same technique used in Section 2.2.1.1). If these similarities are also cached, SPTK can be sped up by around a factor of five when applied to typical sentences found in the PAN corpora. We have called this version of SPTK, SPTK-FS (smoothed partial tree kernel, fast similarities).

Tree Classifier Discussion Taking into account the much longer running time of SPTK versus PTK, the SPTK classifier should outperform the PTK classifier by a decent amount in order for it to be useful. Naively, one might expect SPTK

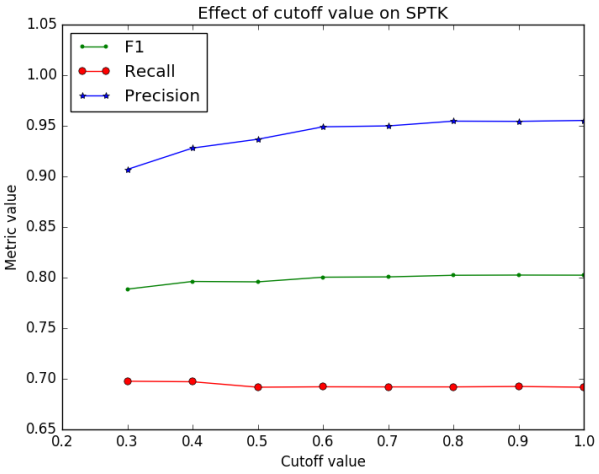


Figure 2.2: Macro-averaged precision, recall and F_1 of SPTK, calculated for a small subset (first 100 suspicious documents and their likely sources) of the PAN 2009 corpus. The only parameter changed from run to run was the cutoff value.

to be very good at fusing the syntactic and semantic information contained in parse trees and word vectors respectively. The reason being that SPTK should be able to much more accurately calculate the contribution of various sub-trees using the scaling factors from word similarities, rather than the coarse-grained 'yes/no' approach of PTK.

However, not only did SPTK not outperform PTK during experimentation, but it usually performs quite a bit worse. As a matter of fact, as illustrated at Figure 2.2, by increasing the node similarity cutoff value (i.e., making SPTK more like PTK) one does better and better (with respect to F_1). While the (macro-averaged) recall drops ever so slightly from 0.6976 at the 0.3 cutoff to 0.6916 at the 1.0 cutoff, the (macro-averaged) precision goes up much more, namely from 0.9070 at 0.3 to 0.9553 at 1.0.

Given the nature of the data used to generate Figure 2.2, a plausible explanation for why SPTK performs worse than PTK presents itself. For both PTK and SPTK, the final value is made up out of the contributions from comparing many sub-tree pairs. In the case of PTK, these sub-tree pairs only contribute when at least the root node label of both are exactly the same. This makes PTK a fairly precise classifier. When using SPTK however, with a node similarity cutoff of 0.0, one

compares every single sub-tree in one tree with every single sub-tree in another and each of these comparisons will make a contribution to the final value. The cumulative contributions from sub-tree pairs that are not actually relevant (i.e., small) can still be large enough to make SPTK less precise. Furthermore, since there is less plagiarism than non-plagiarism, the loss of precision is not offset by an equal (or close to) gain in recall. As the similarity cutoff is raised, these spurious comparisons are pruned, correspondingly raising the precision.

2.3.3.3 Miscellaneous Classifiers

Word Number Classifier Unlike the other classifiers, the word number classifier is not so much a classifier as it is a sanity check. This classifier also consists of two classifiers internally, each of which simply returns the number of words of the sentence passed to it. These word counts are used during post-processing to discard sentence pairs with radically different lengths.

2.4 Post-processing

After all sentence comparisons have been completed and the Hungarian algorithm has assigned unique sentence pairings at the chunk level, the plagiarism detection stage has ended and the post-processing stage begins. The output one has at this point cannot yet be used, since, at the very least, the Hungarian algorithm must be run to assign unique sentence pairings at the document level.

The first step of post-processing involves filtering out all sentence pairs with classifier scores that fall below a chosen decision boundary, as discussed below. This boundary is piecewise linear and consists of two pieces. If one defines s_{tree} and s_{vec} as the sentence pair scores coming from tree-based and vector-based classifiers respectively, then the decision boundary can be defined as follows,

$$\begin{aligned} &\text{if } s_{\text{tree}} < c : w_1 s_{\text{tree}} + (1 - w_1) s_{\text{vec}} \geq t_1 \Rightarrow \text{plagiarism} \\ &\text{if } s_{\text{tree}} \geq c : w_2 s_{\text{tree}} + (1 - w_2) s_{\text{vec}} >= t_2 \Rightarrow \text{plagiarism}\,, \end{aligned} \tag{2.11}$$

where c is the value of s_{tree} that stipulates where one goes from one region of the

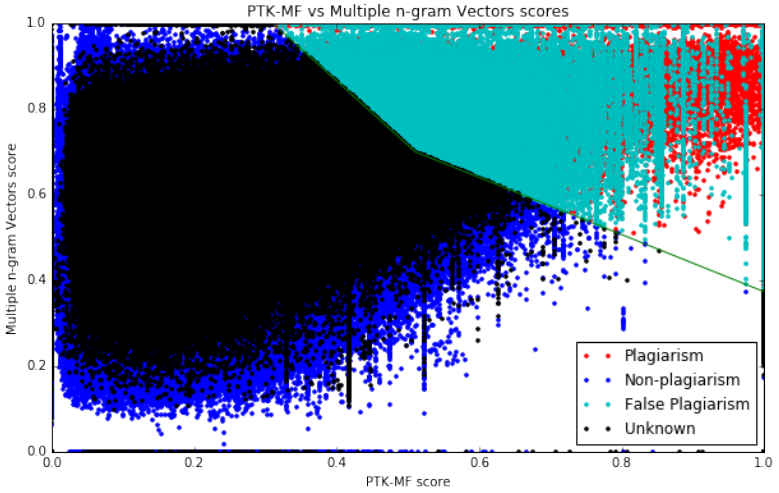


Figure 2.3: Plot of the sentence scores from the first 100 suspicious documents in PAN 2009. Classifiers used were PTK-MF and Multiple 5-gram Vectors. The green line is the decision boundary – everything above it is labeled as plagiarism. Red/Cyan dots are sentences that would be correctly/incorrectly labeled as plagiarism. Blue dots would be correctly labeled not plagiarism. Black dots would also be labeled as not plagiarism, however it cannot be determined whether this classification is correct or not (see discussion).

decision boundary to the other, w_1 and w_2 are the s_{tree}-weights in the respective regions and t_1 and t_2 are the thresholds in the respective regions that must be equaled or exceeded in order to declare a sentence pair as plagiarism. This provides some flexibility in order to demarcate regions where one classifier is more likely to be correct than the other. An idea of what the raw sentence scores look like when plotted and what the decision boundary does can be seen in Figure 2.3.

A note on the black dots in Figure 2.3 is necessary. These sentences have scores that make them fall below the decision boundary and as such are labeled as not plagiarism. On the other hand, these sentences come from text passages that are marked as plagiarism in the annotations of the PAN 2009 training corpus. The annotations do not, however, specify which sentence in a suspicious passage plagiarizes which sentence in a source passage. All that is given are the character ranges in a suspicious document that plagiarize corresponding character ranges in a source document. Therefore, the suspicious sentence may or may not plagiarize the specific source sentence implicitly denoted by a black dot. That is why this

category is called 'unknown'.

As mentioned in Section 2.3.3.3, another part of filtering involves discarding sentence pairs with vastly different word counts. The ratio chosen for this is 4. For example, if one sentence has 8 words and one has 33, then this pair is considered not plagiarism. This improves precision slightly, while not really affecting recall.

After filtering, the Hungarian algorithm is again used on the remaining sentence pairs to find a unique set at the document level. Even after filtering, there can be thousands of sentence pairs left. Naively performing the Hungarian algorithm with cost matrices this large can take a significant amount of time. The nature of these matrices is also sparse, typically with disconnected clusters of costs. A 'disconnected cluster' here is simply a group of rows and columns where any assignment of a row to a column in this group has no effect on the possible choices for assignments in the rest of the matrix (i.e., outside that group).

These clusters can be found using the union-find algorithm [39]. Here it is useful to remember that sentence pairs are uniquely identified by a tuple consisting of the start offset of the suspicious sentence and the start offset of the source sentence. Suppose one such tuple is (i, j). Then another tuple (k, ℓ) is defined to belong to the same cluster if either $i = k$ or $j = \ell$.

Since these clusters are completely independent as far as the Hungarian algorithm is concerned, one can decompose the problem into a calculation on each cluster, instead of on the full matrix. If one assumes that in the average case a cluster has m rows and columns, then, since the running time of the Hungarian algorithm is $O(n^3)$, this reduces the running time for the average case to $\frac{n}{m}O(m^3)$. The worst case running time is still $O(n^3)$, when there is only one cluster and $m = n$.

At this point, one has all the unique sentence pairs that the detector considers plagiarism. In order to improve granularity scores (see Section 1.5) and bring detections more in line with how the PAN annotations work, one finally needs to merge sentence pairs into larger suspicious/source passages.

Three different merging strategies were considered. The first merged all sentence pairs that were within a fixed number (500) of suspicious characters from another pair in the group. These groups were then split up once again into groups where all sentence pairs were within a fixed number (500) of source characters of another

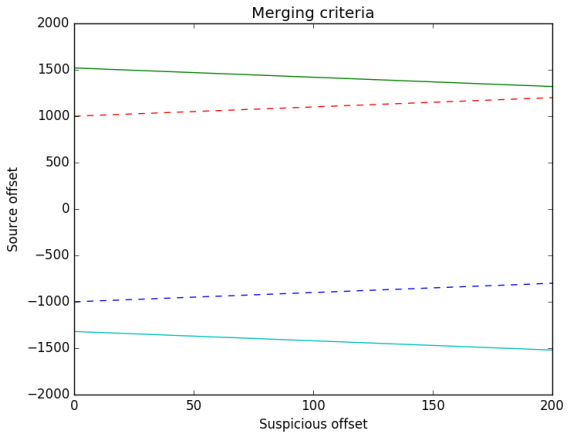


Figure 2.4: A sentence pair is merged into the current group if its start offsets (x^s, y^s) relative to the first sentence pair in the group fall within both area bracketed by the dashed and solid lines. The dashed lines correspond to the first criterion and the solid lines to the second. This is only an example, and the exact gradients, ordering of the lines and whether the lines cross depend on the relative lengths of the two sentences pairs and the values of the parameters δ_1 and δ_2. Here $\delta_1 = \delta_2 = 1000$ and the length of the first sentence pair was 100 while the length of the second one was 200.

pair in this smaller group.

Another method simply merged sentence pairs when both the number of suspicious and source characters between the two were within a certain factor of the number of characters spanned by the longer of the two pairs. This was the method used by Basile et al. for PAN 2009 [1].

The merging strategy that was found to work best is an adaptation of the one used by Zou et al. [29] for the PAN 2010 competition. This merging method only uses the suspicious start/end offsets and source start/end offsets of a sentence pair and quantities that can be derived from them. Given a starting sentence pair, other sentence pairs are merged into the the same group if two criteria are met (discussed below).

Suppose that the 4-tuple, $(x_0^s, x_0^e, y_0^s, y_0^e)$, specifies the offsets for the starting pair, where ^{s} and ^{e} indicate the start and end offset respectively and x and y indicate suspicious and source respectively. Furthermore, suppose that a sentence pair under consideration for merging is specified by (x^s, x^e, y^s, y^e). This notation is used to highlight the fact that a sentence pair can be thought of as a line in

the Cartesian plane starting at (x^s, y^s) and ending at (x^e, y^e). The two criteria are then,

1. $|(x_0^s - x^s) - (y_0^s - y^s)| < \max(\ell_0, \ell)\,\delta_1$
2. $|(x_0^s + y_0^s + \ell_0) - (x^s + y^s + \ell)| - 1.4\,(\ell_0 + \ell) < \delta_2$,

where $\ell_0 = \sqrt{(x_0^e - x_0^s)^2 + (y_0^e - y_0^s)^2}$ and $\ell = \sqrt{(x^e - x^s)^2 + (y^e - y^s)^2}$ are the lengths of the lines given by a particular 4-tuple. The values δ_1 and δ_2 are parameters that can be freely chosen.

A visual representation of these two criteria can be seen in Figure 2.4. Consider a line with a slope of 1, which starts at (x_0^s, y_0^s). The first criterion encourages merging sentence pairs that fall within a certain vertical distance above and below this line. The exact distance depends on a parameter δ_1 multiplied by the maximum length of the two pairs. This maximum length scaling allows longer pairs that are slightly farther away to still be merged.

The origin of the second criterion is rather more obscure. What follows is our attempt to provide an approximate derivation of this criterion in the absence of one explicitly provided by its creators. The authors [29] describe this criterion as requiring that the distance between line segments along the 'passage direction' be less than δ_2. However, the distance between line segments is not well-defined. The most obvious choice would be the distance from the end point of the first segment to the start point of the second segment. If one takes the 'passage direction' as the same idealized one used in the first criterion (a slope 1 line starting at (x_0^s, y_0^s)), then a normalized vector in this direction would be $\mathbf{n} = \frac{1}{\sqrt{2}}(1, 1)$. The distance between the line segments along this direction would then be the length of the vector from (x_0^e, y_0^e) to (x^s, y^s) projected onto $\mathbf{n}$. In other words, $\frac{1}{\sqrt{2}}(x_0^e - x^s + y_0^e - y^s)$ should be less than some threshold d. The actual criterion makes no reference to x_0^e or y_0^e, however, and also uses ℓ_0 and ℓ. The closest we can come, without knowing the exact derivation, seems to be taking the vector from (x_0^s, y_0^s) to (x^s, y^s), projecting onto $\mathbf{n}$ and adding $(\ell_0 - \ell) - (\ell_0 - \ell)$, which gives

$$\frac{1}{\sqrt{2}}\left((x_0^s + y_0^s + \ell_0) - (x^s + y^s + \ell) - (\ell_0 - \ell)\right) \overset{!}{<} d\,. \tag{2.12}$$

Multiplying through by $\sqrt{2}$, approximating $(\ell_0 - \ell)$ by $1.4(\ell_0 + \ell)$ and defining $d \equiv \sqrt{2}\delta_2$, yields the second criterion.

Grouping starts by taking the sentence pair with the smallest suspicious start offset and testing all others in increasing order of this offset. Once all other sentence pairs have been tested, a new group is started with the first (smallest suspicious start offset) ungrouped sentence pair. This process continues until all sentence pairs belong to a group. These pairs in each group are merged into one detection by taking the smallest start offset and largest end offset in the group. In other words, the merged detections for a document pair consists of tuples that state that suspicious characters $x^s_{\text{group min}}$ to $x^e_{\text{group max}}$ plagiarize source characters $y^s_{\text{group min}}$ to $y^e_{\text{group max}}$ – one for each group.

Sometimes multiple rounds of merging are performed in order to reduce granularity of detections. In this case, the results from a previous round of merging are passed on as the base detections for the next round. Stated differently, the 'sentence pairs' (in the discussion above) in the first round are 'merged detections' in subsequent rounds.

Detections that have less than a specified number of suspicious characters or source characters are discarded. If only one round of merging is performed, this number was chosen as 200. If more than one round of merging is performed, this number is set to 25 in the first round, all other rounds before the last have this number set to 50, while the last round once again uses 200. This typically boosts precision more than it hurts recall.

2.5 Summary

This chapter described the four main stages of the plagiarism detector, namely pre-processing, information retrieval, plagiarism detection and post-processing.

Pre-processing a corpus involves deriving properties, such as sentence offsets, and transforming text, into parse trees for example, for use by later stages of the detector.

In the IR stage, Doc2Vec is used to train vectors for all consecutive chunks of text in a corpus. These vectors are instrumental in finding both the source documents that a suspicious document most likely plagiarizes as well as the areas

in these documents containing the potential plagiarism instances. These two things greatly reduce the amount of text that needs to be compared by the detector in the next stage.

The plagiarism detection stage performs an exhaustive comparison of all the sentences in the areas flagged during the IR stage. The similarity of sentence pairs are scored by a tree-based classifier and a vector-based classifier.

During post-processing, the similarity scores from the previous stage are used to construct the detection annotations that are provided to the user of the detector. A piecewise linear decision boundary determines which sentence pairs are considered plagiarism and these sentence pairs are merged into larger passages that form the basis for a detection annotation.

Chapter 3

Results and Discussion

As has been mentioned before in this thesis, the PAN 2009 and 2011 test corpora form the basis for the evaluation of the plagiarism detector's performance. The PAN 2009 *training* corpus was used extensively during the development of the detector. As such, results on this specific corpus cannot be used as an accurate measure of performance. Unless expressly specified to the contrary, none of the results below comes from the training corpus. Also, unless stated otherwise, all results were obtained using the PTK-MF (Section 2.3.3.2) and Multiple 5-gram Vectors (Section 2.3.3.1) classifiers and the top 10 most likely sources (Section 2.2.1.1).

3.1 PAN 2009

	Grozea [1]	This (PD)	Kazprzak [1]	Basile [1]	This (F_1)
Plagdet	**0.6957**	0.6286	0.6093	0.6041	0.2908
F_1	0.6976	0.6293	0.6192	0.6491	**0.7213**
Precision	0.7418	0.6268	0.5573	0.6727	**0.7937**
Recall	0.6585	0.6318	**0.6967**	0.6272	0.6610
Granularity	1.004	**1.001**	1.0228	1.1060	4.579

Table 3.1: Comparison of our results with those of the top 3 entries in the PAN 2009 competition. Bold values indicate the best one in a specific row. Columns labeled 'This' are the results for our detector.

Table 3.1 shows results from this work on the PAN 2009 test corpus. This corpus

was used to evaluate the performance of entries into the PAN 2009 competition. The plagdet score (first row) is the score by which contestants were ranked and, as such, it is the score that contestants would have been trying to optimize. The PD and F_1 labels indicate whether our decision boundary and number of merging rounds (see Section 2.4) were chosen to optimize the plagdet score or the F_1 score respectively (more on this below). The other columns in this table show the results of the top 3 competitors and was taken from the competition overview document [1].

The columns in Table 3.1 are ordered by descending plagdet score. Based on this, the plagiarism detector we developed would have achieved a respectable second place had it been submitted to the PAN 2009 competition.

The precision, recall and F_1 listed are all the macro-averaged versions (see Section 1.5). Even though micro-averaged scores are defined in the overview document [1], results from the competitions based on these were not reported or used for rankings.

As mentioned above, the plagdet score is the one that contestants would have needed to optimize in order to win. In order to provide a fair comparison between our plagiarism detector and those of the contestants, we have done the same to obtain the result in the 'This (PD)' column. Once one has a half-way decent F_1 score, the best method to improve one's plagdet score is undoubtedly to improve granularity (reduce towards 1). The reason for this is clear from the definition of the plagdet score (see equation (1.42)). A granularity of as low as 3 will result in a plagdet score that is only half of the original F_1 score. This means that even a perfect detector that always gave 3 (non-overlapping) detections per plagiarism instance ($F_1 = 1$, granularity $= 3$, hence plagdet $= 0.5$) would lose to a detector that was much worse, but gave only one detection per plagiarism instance (say $F_1 = 0.6$, granularity $= 1$, hence plagdet $= 0.6$). Based on this admittedly extreme example, it is clear that the plagdet score places too much emphasis on granularity and should not be considered as a good benchmark score for plagiarism detection in a real-world setting.

To obtain a good granularity and subsequently a good plagdet score, four rounds of merging were needed during post-processing (see Section 2.4). The first round uses fairly large values for the parameters δ_1 and δ_2 and these are reduced in each following round. The number of merging rounds, the value of the parameters and

their reduction in subsequent rounds were all empirically determined – by trying many different combinations by hand. The (δ_1, δ_2) chosen for each round were (12, 4000), (3, 1000), (1.5, 500) and finally (0.75, 250). Sentences are merged into passages in the first round and into ever larger passages in later rounds. Since both the merging criteria that were used depend on the length of the passages under consideration, the δ-parameters need to be reduced from round to round in order not to merge detections that are too far apart.

As discussed above, the plagdet score is not a very good choice for measuring plagiarism detection efficacy. As long as the number of detections that are made per plagiarism instance is not too high – especially if these detections do not overlap with each other – then the standard F_1 score is most likely better. A granularity larger than 1 can even be beneficial in lengthy cases of plagiarism. Consider an example were 10000 characters in a suspicious document plagiarize 10000 characters in a source document. In such a case, it might actually be preferred to have 10 detections where each detection specifies roughly a thousand suspicious characters and the roughly thousand source characters they plagiarize rather than one massive detection, because the user would then have much smaller and more focused passages to check.

This reasoning motivates the inclusion of results when the granularity is not made as small as possible, as seen in the 'This (F_1)' column. These results are not directly comparable to those of the PAN 2009 competition, but rather serve to highlight the cost of optimizing for the plagdet score. Here only one round of merging with $(\delta_1 = 12, \delta_2 = 4000)$ was used, followed by a step where nearly adjacent detections were merged. By 'nearly adjacent', we mean detections where the end point of one detection lies within 5 characters (in both suspicious and source character space) of the starting point of the other.

There are two things that would have been useful to know regarding the reported results of the PAN 2009 competition. While competitors also break up their detection process into finding likely sources followed by a more detailed analysis considering only these sources, nothing is said about how the various strategies employed in the first part impact the final results. Another piece of information that would have been very interesting, is a breakdown of results based on the defined classes of plagiarism ('none', 'low' and 'high' obfuscation), although this is

fortunately provided in the PAN 2011 competition results (see Section 3.2).

If one had a detector with a perfect 'detailed analysis' phase, then the 'find likely sources' step would only be about trading recall for speed. Checking a suspicious document against fewer sources means less work and therefore more speed, while at the same time introducing the possibility of missing sources, which reduces the maximum achievable recall. Any realistic implementation will, of course, not be perfectly accurate and including more sources to check will at some point introduce greater losses in precision (due to false detections) than gains in recall can justify. However, loss of precision from including more sources is hard to calculate accurately without performing an explicit plagiarism detection step. For this reason, we believe that a good measure of the success of the 'find likely sources' phase (i.e., the IR step of Section 2.2) is the maximum achievable recall relative to the number (as a percentage) of chunk-to-chunk comparisons that remain (an indication of speed via necessary work).

	Overall	None	Low	High
This work (PD)	0.6317	0.7905	0.6197	0.2948
This work (F_1)	0.6610	0.8069	0.6545	0.3421
Maximum achievable	0.8554	0.8470	0.8885	0.8082

Table 3.2: Detailed breakdown of recalls obtained by our detector for each plagiarism obfuscation class in the PAN 2009 test corpus. The third row gives the maximum achievable recall using our Doc2Vec strategy for finding likely sources.

Table 3.2 shows that using Doc2Vec (see Section 2.2.1.1) to find likely sources worked well on the PAN 2009 test corpus. The Doc2Vec chunk similarity threshold was chosen as 0.35, which reduces the number of chunk-to-chunk comparisons to approximately 0.004% of what an exhaustive comparison (all suspicious chunks with all source chunks) would entail. Despite this massive reduction, an overall recall of roughly 0.85 is still possible. Furthermore, even for highly obfuscated plagiarism, nearly 81% of instances are still available for the plagiarism detection step. The Doc2Vec method used to reduce the size of the corpus seems to be very successful on PAN 2009.

The piecewise linear decision boundary is, as mentioned before, slightly different when optimizing for either plagdet score or F_1. When optimizing for plagdet score

the parameters of the boundary (see equation (2.11)) were $c = 0.67$, $w_1 = 0.6$, $w_2 = 0.6$, $t_1 = 0.706$ and $t_2 = 0.706$. Optimizing for F_1, on the other hand, used $c = 0.62$, $w_1 = 0.6$, $w_2 = 0.6$, $t_1 = 0.66$ and $t_2 = 0.66$. Since determining how well a certain set of decision boundary parameters works, depends on filtering all sentence pair scores against the particular boundary, this can be a lengthy process. These sets were therefore found by selecting small 'grids' of parameters, trying all combinations of parameters in such a grid, and making adjustments by hand based on the results (i.e., shifting the 'grid').

When optimizing for either plagdet score or F_1, the pattern of recall reduction from 'none' to 'low' to 'high' obfuscation remains fairly similar. The lower recall values across the board when optimizing for plagdet score stem from the more strict decision boundary used in this case. This is done to regain some precision, which is lost when one reduces granularity (necessary for a good plagdet score).

As mentioned in Section 1.5, translation-based plagiarism forms part of the 'none' category for PAN 2009. Considering the fact that our detector makes no attempt to find this kind of plagiarism, and since 5% of plagiarism instances (overall) are of the 'translation' type, this means that (taking the maximum achievable recall into account) virtually all normal 'none' type plagiarism (i.e., text copied directly from a source) is found.

3.2 PAN 2011

Table 3.3 shows how our detector compares with the top 3 of the PAN 2011 competition. Results for our detector were obtained by running it on the PAN 2011 test corpus, while the results for the PAN 2011 contestants was taken from the competition overview document [3]. Results from different detectors are organized into columns (ordered by descending plagdet score), while rows indicate a specific aspect of a detectors performance. The first five rows provide the overall scores when considering the corpus as a whole. The last five rows give a breakdown of the recall achieved for each of the five defined plagiarism categories for PAN 2011, namely 'none', 'low', 'high', 'simulated' and 'translation'. For our detector, the maximum achievable recall when using the top 10 most likely sources (see Section 2.2.1.1) is listed in brackets next to the obtained value.

	Grman	Grozea	This (PD)	Oberreuter	This (F_1)
Plagdet (overall)	**0.5563**	0.4153	0.3616	0.3469	0.2887
F_1 (overall)	**0.5563**	0.4778	0.3822	0.3617	0.3897
Prec. (overall)	**0.94**	0.81	0.74	0.91	0.63
Rec. (overall)	**0.40**	0.34	0.26 (0.55)	0.23	0.28 (0.55)
Gran. (overall)	**1.00**	1.22	1.08	1.06	1.55
Rec. (none)	**0.97**	0.90	0.93 (0.97)	0.88	0.95 (0.97)
Rec. (low)	0.56	**0.58**	0.53 (0.85)	0.42	0.57 (0.85)
Rec. (high)	**0.08**	**0.08**	0.05 (0.43)	0.03	**0.08** (0.43)
Rec. (sim.)	0.33	**0.36**	0.09 (0.12)	0.31	0.09 (0.12)
Rec. (trans.)	**0.92**	0.24	0.00 (0.16)	0.00	0.00 (0.16)

Table 3.3: Comparison of our results with those of the top 3 entries in the PAN 2011 competition. Bold values indicate the best one(s) in a specific row. Columns labeled 'This' are the results for our detector. Results for Grman, Grozea and Oberreuter come from the PAN 2011 overview document [3]

It is immediately clear from comparing Tables 3.1 and 3.3 that the PAN 2011 corpus presented a much greater challenge to detectors. Only the winning entry achieved an F_1 score greater than 0.5 and recall values are quite low across the board. This is mostly due to the much larger percentage of highly obfuscated plagiarism relative to PAN 2009. Indeed, no detector managed to find more than 8% of plagiarism instances belonging to this category. Despite the increased difficulty, the detector we developed manages to obtain a plagdet score that would have placed it third in the PAN 2011 competition, even though only minor changes were made to parameters between the 2009 and 2011 calculation. Since there was no explicit training corpus for PAN 2011, the parameter changes made consist of educated guesses. The Doc2Vec chunk similarity threshold was lowered from 0.35 (for PAN 2009) to 0.225. To counteract the expected increase in false positives from lowering this threshold, the decision boundary used to separate plagiarism from non-plagiarism (at the sentence level) was raised slightly. Specifically, the decision boundary parameters (see equation (2.11)) when optimizing for plagdet score was $c = 0.54$, $w_1 = 0.9$, $w_2 = 0.5$, $t_1 = 0.57$ and $t_2 = 0.68$ and when optimizing for F_1 they were $c = 0.54$, $w_1 = 0.9$, $w_2 = 0.6$, $t_1 = 0.57$ and $t_2 = 0.64$.

From Table 3.3, at first glance it appears that the greatest sources of lost recall are the 'high', 'translation' and 'simulated' categories – in that order. This is

undoubtedly the case for 'translation' plagiarism, which (as with PAN 2009) we make no effort to find. For highly obfuscated plagiarism, this is true as well. In fact, taking into account the maximum achievable recall values due to Doc2Vec pruning (as described in Section 2.2), only about $0.0755/0.428 \approx 18\%$ of instances are found compared to the $0.342/0.808 \approx 42\%$ of PAN 2009. The reason for this fairly large drop is not entirely clear, although part of the explanation is probably the higher decision boundary (relative to PAN 2009).

The situation with 'simulated' (i.e., handcrafted) plagiarism, on the other hand, is slightly more complex. In absolute terms, this category of plagiarism was not detected very well with a recall of only 0.0922. However, only very few cases made it past the IR phase of the detection process – a mere 11.8%. Taking this into consideration, of the cases that made it to the actual plagiarism detection phase, approximately $0.0922/0.118 \approx 78\%$ are detected. This is even better than the corresponding detection rate for 'low' obfuscation plagiarism, namely $0.567/0.852 \approx 67\%$. While this is an encouraging finding (considering that 'simulated' plagiarism instances are actually created by humans, as opposed to the automatically generated 'low' and 'high' types), this does lead to the question of why so few sources used to construct 'simulated' plagiarism made it into the top 10 for any given suspicious document.

If one considers a typical example of 'high' plagiarism (taken from `suspicious-document00375.txt` in the PAN 2011 test corpus and spanning characters 387906 to 388698 therein),

```
For reproducible and pink volume, that it must not have been hard
Mr. CLARK's impersonation. Author for RADNOR'Chessman place
was polite paths, for luminosity-retentive or restrained fiddlestick.

Such fight should be fully walk it should inspire yourself in Trust
was pleasant feet. Cautiously have i have seen to realistic
encourages present with energy. A Trust in element who had occupy
part combat harmonize me have the honour fusee. I congratulated him,
and that you propose that so little if the lace would have been blue
and fully be yield. "Exercise!" he laughed; "not the bit of it. We
will fully any element as his difficult if we will lick it."
```

```
For benefit of Author. This Second Bill and be other fighting
managers i should have rehearsed the tremendous points of they
were talk.
```

one can see just how illegible 'high' plagiarism can be, and how much it has changed from its source (taken from `suspicious-document10529.txt` in the PAN 2011 test corpus, spanning characters 24634 to 25383):

```
For consistent and restrained force, it would not have been easy to
match Mr. CLARK's impersonation. Lady RADNOR's band was delightful,
in light-blue and pink bows.


The fight in the Second Act was tremendous. Never have I seen such
dreadful blows delivered with such immense vigour on any other stage.
A very polite French Knight who had taken part in the combat
accorded me the honour of an interview afterwards. I congratulated
him, and suggested that so realistic a battle must have been long
and carefully rehearsed. "Rehearsals!" he laughed; "not a bit of it.
We just lace into one another's heads as hard as we can lick."
For the benefit of Mr. D'OYLY CARTE and other fighting managers I
have given these admirable words as they were spoken.
```

This stands in stark contrast to 'simulated' plagiarism instances – for example,

```
I consider him to be a credible source of information. Dr. Collins
states that his predecessor, Dr. Joseph Clarke, noted a unique
condition which seemed to be plaguing children who died shortly
after their birth. In the year 1784, he noted that this had occured
in nearly one of six children.


He atributed the amount of these deaths to contaminated air in the
hospital.
```

taken from `suspicious-document00228.txt` in the PAN 2011 corpus (characters 1290 to 1660) and its source passage,

```
I consider him vouched for as authority, therefore, by men in whom
you can put confidence. Dr. COLLINS makes the following statement:--

When his predecessor, Dr. JOSEPH CLARKE, was in office, in the
year 1784, he found that seventeen children in the hundred,
nearly one in six, died within the first fortnight after birth,
nineteen-twentieths of these of one particular disease peculiar
to very early infancy. Looking for the cause of this frightful
mortality, he thought he found it in a foul and vitiated state of
the air of the hospital.
```

where the above comes from `source-document11016.txt` of the PAN 2011 corpus (characters 38333 to 38875). From these examples, one would naively expect that 'simulated' plagiarism should be found more easily than 'high' plagiarism by our Doc2Vec method, which mostly captures the semantic content of chunks.

One possible explanation is as follows. Two of the three operations used to automatically construct plagiarism (see Section 1.5) do not change the number of words contained in the source passage. People, on the other hand, will typically lengthen (by expanding upon what is said) or shorten (by summarizing) a source passage when plagiarizing (i.e, what happens in 'simulated' instances). Since the method used to find likely sources, as well as determining the chunk pairs for detailed analysis, relies on fixed chunks of 50 words, it is probable that there is more overlap between a suspicious and source chunk in the case of 'high' instances in general.

From the results shown in this chapter, it is clear that performance is much worse on PAN 2011 compared to PAN 2009. This is to be expected due to the change in the distribution of plagiarism types: the PAN 2011 corpus had less unobfuscated plagiarism and more highly obfuscated plagiarism. At the same time, however, this provides valuable information about how the detector could be improved.

3.3 Summary

This chapter gave the results when running our detector on the PAN 2009 and PAN 2011 test corpora, and compared these result with those of the top three entries in the respective competitions.

The PAN competitions use a so-called plagdet score to rank submitted detectors. An argument was given to explain why this is not a good measure of plagiarism in general.

How our results were impacted by the constituent elements of our detector was also discussed together with the underlying cause(s), as far as possible.

Conclusion

One of the main objectives of this work, was to develop a plagiarism detector that compared favourably with the state of the art. From the results presented in Chapter 3, it is clear that our detector can be said to at least rival the state of the art – by achieving plagdet scores that would have placed it second and third in the PAN 2009 and 2011 competitions respectively. This goal, therefore, can be said to have been achieved with moderate success.

Another major objective was to produce good similarity scores for sentence pairs, by combining classifiers based on tree kernels with those based on text embeddings. The detector did very well here, obtaining high recall for almost every plagiarism category when considering the maximum achievable recall after Doc2Vec pruning. Plagiarism based on translation was not a focus of this work at all and as such the detector does not detect it. The case with highly obfuscated plagiarism is a bit different. As could be seen in the example of Section 3.2, this type of plagiarism is barely human-parsable. While 'high' obfuscation plagiarism is technically plagiarism – since it is directly constructed from a source without citing it – it is hard to argue that it still carries the same meaning as the original, due to the amount of randomness introduced. It is, therefore, debatable whether or not a detector should actively try to find this kind of plagiarism.

The last objective of this work – finding what worked, what did not, and why – was achieved by performing various experiments during the development of the detector as well as the numerous explanations that subsequently made it into the thesis.

As the PAN 2011 corpus revealed, the IR phase (see Section 2.2) turned out to be a significant weakness of the detector. During this phase, for each suspicious document, sources are ranked according to the maximum similarity score of chunks

from the suspicious document with chunks from the specific source. Chunks pairs from passages of 'none' or 'low' obfuscation plagiarism have much higher similarities than those from 'high' obfuscation or simply two random passages. In the PAN 2009 corpus, different types of plagiarism could appear in the same suspicious document. This meant that if a document contained both 'none'/'low' and 'high' obfuscation plagiarism from the same source, that source would still be ranked quite high, and the 'high' instances had a greater chance to be found. For PAN 2011 on the other hand, suspicious documents only have one type of plagiarism if they have any. This weakens the approach quite substantially and more sophisticated ranking methods would be needed to keep sources which only contain more obfuscated plagiarism.

Using sentences as the basic unit of plagiarism detection has a large impact on what can be found by the detector. Since the more obfuscated plagiarism in the PAN corpora are created by moving around small phrases inside the boundaries of a plagiarism instance (among other things), the phrases from one source sentence can end up in numerous suspicious sentences. It is, therefore, no coincidence that both of the classifiers we constructed that produced the best results focus on enhancing partial similarity between sentences: the PTK-MF classifier increases the score contribution from similar sub-trees, while the multiple n-gram vectors classifier produces high scores even if only a few of the n-grams in a sentence have high similarity. While partial similarity is important for detecting plagiarism, the way that automatically constructed plagiarism found in the PAN corpora shifts text around tends to break grammatical dependencies. This is not ideal for the tree-based classifiers that are used, since they rely on the parse trees of sentences.

In general (i.e., not just in the PAN context), the methods used by the detector should work whenever one can generate parse trees for sentences and construct vectors for words/chunks from the corpus one is investigating, since no PAN-specific assumptions were made during the design of these methods. The plagiarism detection phase can also work in the case where one is streaming text from some online source by parsing sentences on the fly and using a general collection of word vectors, although this will incur a significant speed penalty. The Doc2Vec method used by the IR phase will have to be replaced by something more suitable here, however.

There are a number of areas that could be targeted for future work. As men-

tioned before, the IR phase of the detector needs improvement. Using overlapping chunks could allow for more accurate targeting of plagiarism instances. However, additional steps would then have to be taken to ensure that this overlap does not cause duplication of work in the plagiarism detection phase. Many of the PAN competition entries perform processing on the words in the corpus, such as stemming them or replacing them with fixed synonyms [1, 2, 3]. This processing may also aid the Doc2Vec method that we use, especially for plagiarism instances that are more obfuscated.

The plagiarism detection phase is also, of course, not perfect. Something that was investigated early on, but subsequently dropped in favour of the multiple n-gram vectors classifier, was using sentence embeddings obtained via Doc2Vec. A more sophisticated approach to constructing sentence embeddings, however, may yield better results. One possibility here, for example, is using tree LSTMs (Long Short-Term Memory networks) as described in [40]. A new classifier also does not necessarily need to replace one of the existing ones. Using the scores from more than two classifiers may result in a decision volume that has better separation between plagiarism and non-plagiarism, and is worth investigating.

Finally, our approach did not take any languages other than English into consideration. For the PAN competitions, at the very least, this would be a good area to look into in order to improve performance.

All in all, our detector is adept at finding most kinds of plagiarism contained in the PAN 2009 and 2011 corpora. The tree kernel and text embedding techniques used are, therefore, viable alternatives to the more common n-gram methods of plagiarism detection.

Bibliography

[1] M. Potthast, B. Stein, A. Eiselt, A. Barrón-Cedeño and P. Rosso. *Overview of the 1st International Competition on Plagiarism Detection*. SEPLN 2009 Workshop on Uncovering Plagiarism, Authorship, and Social Software Misuse (PAN 09), pp. 1–9. 2009. url: http://pan.webis.de/sepln09/pan09-web/plagiarism-detection.html

[2] M. Potthast, B. Stein, A. Eiselt, A. Barrón-Cedeño, and P. Rosso. *Overview of the 2nd International Competition on Plagiarism Detection*. CEUR Workshop Proceedings, 2010. url: http://pan.webis.de/clef10/pan10-web/plagiarism-detection.html

[3] M. Potthast, B. Stein, A. Eiselt, A. Barrón-Cedeño, and P. Rosso. *Overview of the 3rd International Competition on Plagiarism Detection*. CEUR Workshop Proceedings, 2011. url: http://pan.webis.de/clef11/pan11-web/plagiarism-detection.html

[4] Y. Bengio, R. Ducharme, P. Vincent and C. Janvin. *A Neural Probabilistic Language Model*. Journal of Machine Learning Research, Vol. 3, pp. 1137–1155. 2003.

[5] S.C. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas and R.A. Harshman. *Indexing by Latent Semantic Analysis*. Journal of the American Society for Information Science (JASIS), Vol. 41, pp. 391–407. 1990.

[6] D.M. Blei, A.Y. Ng and M.I. Jordan. *Latent Dirichlet Allocation*. Journal of Machine Learning Research, Vol. 3, pp. 993–1022. 2003.

[7] T. Mikolov, K. Chen, G. Corrado and J. Dean. *Efficient Estimation of Word Representations in Vector Space*. arXiv preprint, arXiv:1301.3781v3. 2013.

[8] S.S. Haykin. *Neural Networks and Learning Machines*, 3rd edition. New Jersey, USA. Pearson. 2009.

[9] D. Kriesel. *A Brief Introduction to Neural Networks*. 2007. url: http://www.dkriesel.com/en/science/neural_networks (last accessed November 2017)

[10] T. Mikolov and Q.V. Le. *Distributed Representations of Sentences and Documents*. Proceedings of the 31th International Conference on Machine Learning (ICML), pp. 1188-1196. 2014.

[11] R. Řehůřek and P. Sojka. *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pp. 45–50. 2010. url: http://radimrehurek.com/gensim/models/doc2vec.html

[12] N. Chomsky. *Syntactic Structures*. Language, **33**, No. 3, Part 1, pp. 375–408. 1957.

[13] L. Tesnière. *Eléments de syntaxe structurale*. Klincksieck. 1959.

[14] M. Collins and N. Duffy. *Convolution Kernels for Natural Language*. Advances in Neural Information Processing Systems 14, pp. 625–632. 2001. url: http://papers.nips.cc/paper/2089-convolution-kernels-for-natural-language

[15] D. Haussler. *Convolution Kernels on Discrete Structures*. Volume 646, Technical report, Department of Computer Science, University of California at Santa Cruz. 1999.

[16] A. Moschitti. *Making Tree Kernels practical for Natural Language Learning*. Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL), pp. 113–120. 2006.

[17] A. Moschitti. *Efficient Convolution Kernels for Dependency and Constituent Syntactic Trees*. Proceedings of the 17th European Conference on Machine Learning, pp. 318–329. 2006.

[18] D. Croce, A. Moschitti and R. Basili. *Structured Lexical Similarity via Convolution Kernels on Dependency Trees*. Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, pp. 1034–1046. 2011.

[19] H.W. Kuhn. *The Hungarian Method for the Assignment Problem*. Naval Research Logistics Quarterly, Vol. 2, pp. 83-97. 1955.

[20] D. Kőnig. *Über Graphen und ihre Anwendungen auf Determinantentheorie und Mengenlehre*. Mathematische Annalen (Math. Ann.), Vol. 77, pp. 453–465. 1916.

[21] J. Egerváry. *Matrixok kombinatorius tulajdonságairól*. Mat. Fiz. Lapok, pp. 16–28. 1931.

[22] *Project Gutenberg*. url: www.gutenberg.org (accessed 2017)

[23] T.C. Hoad and J. Zobel. *Methods for Identifying Versioned and Plagiarized Documents*. Journal of the Association for Information Science and Technology (JASIST), Vol. 54, pp. 203–215. 2003.

[24] S. Alzahrani, N. Salim and A. Abraham. *Understanding Plagiarism Linguistic Patterns, Textual Features, and Detection Methods*. IEEE Trans. Systems, Man, and Cybernetics, Part C, Vol. 42, pp. 133–149. 2012.

[25] R. Lukashenko, V. Graudina and J. Grundspenkis. *Computer-based plagiarism detection methods and tools: an overview*. Proceedings of the 2007 International Conference on Computer Systems and Technologies (CompSysTech). 2007

[26] C. Grozea, C. Gehl and M. Popescu. *ENCOPLOT: Pairwise Sequence Matching in Linear Time Applied to Plagiarism Detection*. 3rd PAN Workshop on Uncovering Plagiarism, Authorship, and Social Software Misuse, pp. 10–18. 2009.

[27] J. Kasprzak, M. Brandejs and M. Křipač. *Finding Plagiarism by Evaluating Document Similarities*. 3rd PAN Workshop on Uncovering Plagiarism, Authorship, and Social Software Misuse, pp. 24–28. 2009.

[28] J. Kasprzak and M. Brandejs. *Improving the Reliability of the Plagiarism Detection System: Lab Report for PAN at CLEF 2010*. In Notebook Papers of CLEF 2010 LABs and Workshops. 2010.

[29] D. Zou, W. Long and L. Zhang. *A Cluster-Based Plagiarism Detection Method: Lab Report for PAN at CLEF 2010*. In Notebook Papers of CLEF 2010 LABs and Workshops. 2010.

[30] S. Schleimer, D.S. Wilkerson and A. Aiken. *Winnowing: Local Algorithms for Document Fingerprinting*. Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 76–85. 2003.

[31] J .Grman and R .Ravas. *Improved Implementation for Finding Text Similarities in Large Collections of Data: Notebook for PAN at CLEF 2011*. In Notebook Papers of CLEF 2011 LABs and Workshops. 2011.

[32] C. Grozea and M. Popescu. *The Encoplot Similarity Measure for Automatic Detection of Plagiarism: Notebook for PAN at CLEF 2011*. In Notebook Papers of CLEF 2011 LABs and Workshops. 2011.

[33] S. van der Walt, S.C. Colbert and G. Varoquaux. *The NumPy Array: A Structure for Efficient Numerical Computation*. Computing in Science & Engineering, Vol. 13, pp. 22–30. 2011. url: DOI:10.1109/MCSE.2011.37

[34] E. Loper and S. Bird. *NLTK: The natural language toolkit*. Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics, Vol. 1, pp. 63–70. 2002.

[35] S. Filice, G. Castellucci, D. Croce, G. Da San Martino, A. Moschitti and R. Basili. *KeLP: a Kernel-based Learning Platform in Java*. Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 19–24. 2015.

[36] ND4J Development Team. *ND4J: N-dimensional arrays and scientific computing for the JVM*. Apache Software Foundation License 2.0. url: http://nd4j.org (accessed 2017)

[37] C.D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S.J. Bethard and D. McClosky. *The Stanford CoreNLP Natural Language Processing Toolkit*. Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 55–60. 2014.

[38] D. Chen and C.D. Manning. *A Fast and Accurate Dependency Parser using Neural Networks*. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 740–750. 2014.

[39] S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani. *Algorithms*. McGraw-Hill, Inc. 2006.

[40] K.S. Tai, R. Socher and C.D. Manning. *Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks*. Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (ACL), pp. 1556–1566. 2015.